UCRL-ID-117524

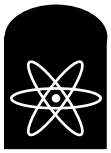
Survey of Industry Methods for Producing Highly Reliable Software

Prepared by

J. Dennis Lawrence
Warren L. Persons

Prepared for

U.S. Nuclear Regulatory Commission



FESSP
Fission Energy and Systems Safety Program

Lawrence Livermore National Laboratory

This work was supported by the United States Nuclear Regulatory Commission under a Memorandum of Understanding with the United States Department of Energy, and performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

Survey of Industry Methods for Producing Highly Reliable Software

Manuscript date: August 29, 1994

Prepared by
J. Dennis Lawrence
Warren L. Persons
Lawrence Livermore National Laboratory
7000 East Avenue
Livermore, CA 94550

Prepared for
U.S. Nuclear Regulatory Commission

ABSTRACT

The Nuclear Reactor Regulation Office of the U.S. Nuclear Regulatory Commission is charged with assessing the safety of new instrument and control designs for nuclear power plants which may use computer-based reactor protection systems. Lawrence Livermore National Laboratory has evaluated the latest techniques in software reliability for measurement, estimation, error detection, and prediction that can be used during the software life cycle as a means of risk assessment for reactor protection systems.

One aspect of this task has been a survey of the software industry to collect information to help identify the design factors used to improve the reliability and safety of software. The intent was to discover what practices really work in industry and what design factors are used by industry to achieve highly reliable software. The results of the survey are documented in this report.

Three companies participated in the survey: Computer Sciences Corporation, International Business Machines (Federal Systems Company), and TRW. Discussions were also held with NASA Software Engineering Lab/University of Maryland/CSC, and the AIAA Software Reliability Project.

CONTENTS

1. Introduction 1
   1.1. Purpose 1
   1.2. Scope 1
   1.3. Report Organization 1
   1.4. Acknowledgments 1
2. Industry Use of Design Factors 2
   2.1. The Most Important Design Factors 2
      2.1.1. Company Commitment 2
      2.1.2. Company Experience 2
      2.1.3. Configuration Management 2
      2.1.4. Independent Verification, Validation, ... 2
      2.1.5. Life Cycle 2
      2.1.6. Metrics 2
      2.1.7. Process Improvement 3
   2.2. Additional Important Design Factors 3
      2.2.1. Cost and Schedule 3
      2.2.2. Risk Management 3
      2.2.3. Use of Standards 3
      2.2.4. Root-Cause Determination 3
      2.2.5. Design for Testing 3
      2.2.6. Certification 3
      2.2.7. Low Turnover 3
      2.2.8. Reading 3
      2.2.9. Document Management 3
      2.2.10. Importance of "-ilities" 3
3. Computer Sciences Corporation 4
   3.1. Introduction 4
   3.2. Practical Application of Software Testing 4
   3.3. Detailed Discussion of Testing 4
      3.3.1. Level R Testing 4
      3.3.2. Level 0 Testing 6
      3.3.3. Level 1 Testing 7
      3.3.4. Level 2 Testing 7
      3.3.5. Level 3 Testing 7
      3.3.6. Level 4 Testing 8
   3.4. Summary of Testing Costs 9
   3.5. Configuration Management 10
4. International Business Machines 12
   4.1. Introduction 12
   4.2. Key Measures to Improve Product Value 12
   4.3. Software Process Improvement 14
      4.3.1. Development Process Lessons Learned 14
      4.3.2. Requirements Management Lessons Learned 16
      4.3.3. Independent Verification Lessons Learned 16
      4.3.4. Quality Management Lessons Learned 16
   4.4. Software Reliability 17
   4.5. Aspects of Software Development 18
      4.5.1. Cost and Quality Planning 18
      4.5.2. Software Requirements Process 20
      4.5.3. Error Cause Analysis and Defect Prevention 23
      4.5.4. Obstacles to Highly Reliable Software 24

5. TRW 24
5.1. Introduction 24
5.1.1. Software Engineering Process Groups 24
5.1.2. Highly Reliable Software 25
5.1.3. Certification of Safety-Critical Software 26
5.2. The AWIS Project 27
5.3. The UNAS Project 27
5.4. Discussion 31
5.4.1. Obstacles to Producing Highly Reliable Software 31
5.4.2. Success in Highly Reliable Software Production 32
6. AIAA Software Reliability Working Group 33
6.1. Background 33
6.2. AIAA Software Reliability Database 33
6.3. Software Reliability Tools Survey 34
6.4. Software Reliability Models 34
6.5. AIAA Software Reliability Recommended Practices 35
7. University of Maryland and NASA Software Engineering Laboratory 35
7.1. Document Reading 35
7.2. Life Cycle Comments 36
7.3. Miscellaneous Comments 37

FIGURES

Figure 3-1. Example of an Independent Test Organization 5
Figure 3-2. Level R Testing 5
Figure 3-3. Example of a Level 3 Test Design 8
Figure 3-4. Percentage of Errors Found at Different Levels of Testing 9
Figure 4-1. On-Board Shuttle Software Error Measurements 13
Figure 4-2. Introduction of Process Improvements 15
Figure 4-3. Software Life Cycle Costing Methodology 19
Figure 4-4. Software Criticality vs. Cost for Early Development Phase of Large Software Systems 21
Figure 4-5. Cost of an Assumption Error in the Requirements Phase 22
Figure 4-6. Requirements Analysis Process Steps 22
Figure 5-1. The Evolutionary Development Environment and Product 28
Figure 5-2. Required Assets and Attributes 29
Figure 5-3. Software System Layers 30
Figure 5-4. Quality Improvements with UNAS 32
Figure 7-1. Life Cycle Phase and Construction Analysis Activities 36
Figure 7-2. Estimated Optimal Size for Modules 37

ABBREVIATIONS

AIAA	American Institute of Aeronautics and Astronautics
ALOC	Ada Lines of Code
AWIS	Army WWMCCS Information System
BMD	Ballistic Missile Division (TRW)
CASE	Computer Assisted Software Engineering
CDR	Conceptual Design Review
CM	Configuration Management
COCOMO	Constructive Cost Model
COS	Committee on Standards (AIAA)
COTS	Commercial Off-The-Shelf
CSC	Computer Sciences Corporation
DBMS	Database Management System
DM	Data Management
DoD	Department of Defense
EDP	Evolutionary Development Paradigm
ELOC	Executable Lines of Code
FAA	Federal Aviation Authority
FSC	Federal Systems Company (IBM)
I&C	Instrumentation and Control
IBM	International Business Machines
IC	Integrated Circuit
IEEE	Institute of Electrical and Electronics Engineers
KLOC	Thousand Lines of Code
LLNL	Lawrence Livermore National Laboratory
NASA	National Aeronautics and Space Administration
NRC	Nuclear Regulatory Commission
NRR	Nuclear Reactor Regulation
OBS	On-Board Shuttle
OOD	Object-Oriented Design
OOP	Object-Oriented Programming
PC	Personal Computer
QA	Quality Assurance
R&D	Research and Development
RA	Requirements Analysis
RADC	Rome Air
RPS	Reactor Protection System
SBOS	Space-Based Observation Systems (AIAA)
SCMP	Software Configuration Management Plan
SED	Systems Engineering Division (CSC)
SEI	Software Engineering Institute
SEL	Software Engineering Laboratory (NASA)
SEPG	Software Engineering Process Group
SLOC	Source Lines of Code
SMERFS	Statistical Modeling and Estimation of Reliability Functions for Software
TQM	Total Quality Management
UNAS	Universal Network Architecture Services (TRW)
V&V	Verification and Validation
WWMCCS	World Wide Military Command and Control System

SURVEY OF INDUSTRY METHODS FOR PRODUCING HIGHLY RELIABLE SOFTWARE

1. INTRODUCTION

1.1. Purpose

The Nuclear Reactor Regulation Office of the U.S. Nuclear Regulatory Commission (NRC/NRR) is charged with (among other responsibilities) assessing the safety of new instrument and control (I&C) designs for nuclear power plants which may use computer-based reactor protection systems (RPS). One task carried out by the Lawrence Livermore National Laboratory (LLNL) in support of NRC activities has been to evaluate the latest techniques in software reliability for measurement, estimation, error detection, and prediction that can be used during the software life cycle as a means of risk assessment for reactor protection systems.

One aspect of this task has been a survey of the software industry to collect information to help identify the design factors used to improve the reliability and safety of software. The intent was to discover what practices really work in industry and what design factors are used by industry to achieve highly reliable software. The results of the survey are documented in this report.

Three companies participated in the survey:

- Computer Sciences Corporation
- International Business Machines, Federal Systems Company
- TRW.

Discussions were also held with two other organizations:

- NASA Software Engineering Lab/University of Maryland/CSC
- AIAA Software Reliability Project.

The results of the survey and discussion were reviewed by personnel from another company, who provided comments on the applicability of the results to other industry sectors.

1.2. Scope

This report documents the discussions with the companies and organizations that participated in the survey. No judgments are made about this information—the report is limited to organizing and presenting the opinions expressed by each company and organization.

1.3. Report Organization

The report contains six sections in addition to this introduction. Section 2 provides a summary of what was learned during the project. It combines the design factors considered important to the companies into two lists. Sections 3–7 provide summaries of the information received from each company and organization. These summaries have been reviewed by the companies visited for completeness and accuracy, and to provide assurance that no proprietary information is included.

Wording in Sections 3–7 frequently takes the form "Company X says ..." or "Company Y believes ..." In each case this should be interpreted as meaning that "the people interviewed at Company X said ..." or "the people interviewed at Company Y believe ..." The opinions expressed are those of the individuals, and may or may not constitute company policy.

1.4. Acknowledgments

Many individuals in the companies visited helped us understand the software development methods used by the companies, and what design factors the companies considered important in the production of highly reliable software. We particularly appreciate the efforts of the following individuals:

For AIAA: George Stark.

For CSC: Victoria Davison and Gail Phipps.

For IBM: Ted Keller, Barbara Kolkhorst, Earl Lee, and Kyle Rone.

For SEL: Victor Basili.

For TRW: Dan Crandall, Mark Gerhardt, James Pearson, Winston Royce, Walker Royce, Stash Vann, and George Ziff.

The authors also acknowledge the contributions of the U.S. NRC Technical Monitor, Mr. John M. Gallagher, in the formulation of and active participation in this task.

2. INDUSTRY USE OF DESIGN FACTORS

Each of the companies surveyed was asked for the most important factors which, in their opinion, are necessary to produce highly reliable software. There was considerable agreement on these factors—seven were identified as the most important. An additional ten factors were identified as important, but not as important as the first seven. Each group of factors is described briefly in the remainder of this section. No ordering is implied within either of the two groups.

2.1. The Most Important Design Factors

2.1.1. Company Commitment

There must be a top-to-bottom commitment to quality work. In particular, middle management must understand, share, and enforce this commitment. If the company is not primarily a software company, then the company's top management must understand the peculiarities of developing high-integrity software, and must be committed to spending the resources required to "make it work."

2.1.2. Company Experience

Organizations responsible for developing trusted software[1] must have considerable experience in successfully producing high-integrity software. The methods used to develop trusted software must become a part of the "corporate culture," and that requires years to develop and mature. Experimentation with new methods, trial and error, and learning from mistakes were all emphasized as key components of the company culture and experience.

2.1.3. Configuration Management

Configuration management must be understood and practiced, since there is no other way to keep track of what software components are actually being developed, tested, and deployed. Changes to requirements must be formally approved, their implications in the software development process must be documented, and the implementation of the changes must be tracked and verified.

2.1.4. Independent Verification, Validation, and Testing

Verification and validation (V&V) and testing must be carried out by an organization which does not report to or depend upon the development organization. The V&V organization may be an independent organization within the same company, or may be a different company; provided that the independence exists and is enforced, this difference does not appear to matter.

2.1.5. Life Cycle

Software development must be based on a company-accepted and understood life cycle. The particular life cycle that is used is not important; the existence of some life cycle is important. The actual life cycle depends on contract stipulations and company experience.

2.1.6. Metrics

All work should be measured, and the measurements must be understood and used. This includes technical work, V&V work, and management work. The actual measurements that are used vary among the companies, and reflect their different cultures and goals. All companies emphasized the importance of measuring both process and product, and using the results to track development and to improve the software development process.

[1] Software is considered to be trusted if the failure of the software creates a risk to safety or security.

2.1.7. Process Improvement

The development organization must actively work to improve itself. Mistakes must be analyzed, root causes must be determined, and the development process must be changed to reflect "lessons learned." The analysis must be done in a non-threatening way, and change must be introduced deliberately and carefully, and paced so as not to be disruptive. Changes to the development process require about two years from their introduction until they become completely integrated into the organization's normal development practices. Some changes take less than two years to be integrated into company practice; others, more.

2.2. Additional Important Design Factors

2.2.1. Cost and Schedule

Management must understand and be able to control cost, schedule, and staffing. This must precede the understanding and control of other quality factors; if cost and schedule cannot be measured and controlled, there is little hope of controlling reliability and other forms of quality.

2.2.2. Risk Management

Management and engineers must understand the concept of risks, and must be able to detect, analyze, and manage development risks.

2.2.3. Use of Standards

Corporate standards should exist for software development, must be understood by all corporate personnel, must be used, and must be enforced. Standards at the corporate level are generally non-specific, but project standards must be specific to the needs of the project and consistent with one another.

2.2.4. Root-Cause Determination

Recurring defects should initiate determination of root causes, and cause changes in organizational behavior to eliminate or control the root causes of defects.

2.2.5. Design for Testing

Requirements specifications and design specifications need to be written so that they may be tested.

2.2.6. Certification

Safety-critical systems should be certified before being deployed. In particular, the development organization managers should be required to sign that the software is ready for use, and should accept personal responsibility for preventable errors.

2.2.7. Low Turnover

Personnel attrition rates should be less than about 5% per year, to prevent loss of corporate memory. Unstable development teams tend to create unreliable software products. Also, high turnover generally indicates poor management policies or actions. Low turnover is necessary both at the technical and managerial levels.

2.2.8. Reading

Errors in requirements, design, and code can frequently be found by reading the specifications or code. Reading for errors is a skill which must be taught and practiced. For best results, reading should be by various specialists: tester, developer, subject area expert, maintainer, etc.

2.2.9. Document Management

Documentation must be planned, produced, analyzed, controlled, and managed according to company standards.

2.2.10. Importance of "-ilities"

The various "-ilities" are important, and cannot be ignored. The following list, taken from a variety sources, is intended to be complete. Any particular projects will need to be selective and set priorities: Access control, accuracy, auditability, commonality, communicativeness, completeness, conciseness, (internal and external) consistency, correctness, efficiency (performance), error tolerance, expandability, flexibility, generality, independence, integrity, interoperability, (internal) instrumentation, maintainability, modularity, operability, portability, reliability, reusability, robustness,

security, self- descriptiveness, simplicity, testability, traceability, training, usability.

3. COMPUTER SCIENCES CORPORATION

3.1. Introduction

Computer Science Corporation (CSC) is a $2.5B corporation, of which $1.1B is from federal government contracts. There are 25,000 employees, including 15,000 professionals. The corporation is divided into a number of groups, one of which, the Systems Group, is responsible for federal contracts.

The Systems Group is itself divided into a number of divisions. Personnel from the Systems Engineering Division (SED) were interviewed for this task. Another division, the Systems Science Division, works with the NASA Software Engineering Laboratory.

The Systems Group has learned three important general lessons about producing highly reliable software:

1. It is necessary to have corporate standards for the various software development activities. For programming languages, there must be a separate style standard for each language. The corporate standards must be enforced.

2. Software must be designed and built so that it can be tested.

3. The testing organization must be independent of the development organization.

The discussions with SED were primarily about software testing.

3.2. Practical Application of Software Testing

CSC/SED tried several different methods for improving the quality of delivered software. The only method that has been found to work consistently is to have an independent software test group, whose sole job is testing. This group is responsible for the quality of the software after it leaves the testing activity. The test group must be in place from the first day of the project—that is, during project planning. The best and brightest software engineers are located in this group. The group must be independent—it does not report to the manager responsible for software development. Instead, the test manager and the development manager both report to an overall project manager. SED's preferred organization chart is shown in Figure 3-1.

It is critical to keep a low turnover rate—less than 2.5%. An auditor should look at turnover rate; if it is high, the auditor can conclude that the system under development is in trouble.

3.3. Detailed Discussion of Testing

CSC/SED has been developing a testing method for the past fifteen years. It uses a scheme with multiple levels of testing. These levels are as follows:

R Requirements Testing

0 Design and Code Walkthroughs

1 Unit Testing

2 Component Testing

3 Program and Subsystem Testing

4 System Testing

3.3.1. Level R Testing

Level R testing, the first level of assurance, consists of software requirements validation. SED has only recently begun using this form of validation The objective is to carry out various forms of analysis to assure the developers that the right product is being developed in the right way. Four forms of analysis may be used, depending on the type of problem being solved. These are mathematical modeling, analytical modeling, simulations, and rapid prototyping.

As shown in Figure 3-2, the level R testing assumes the existence of a system requirements specification and a high-level architectural design, and relates the software requirements to that design.

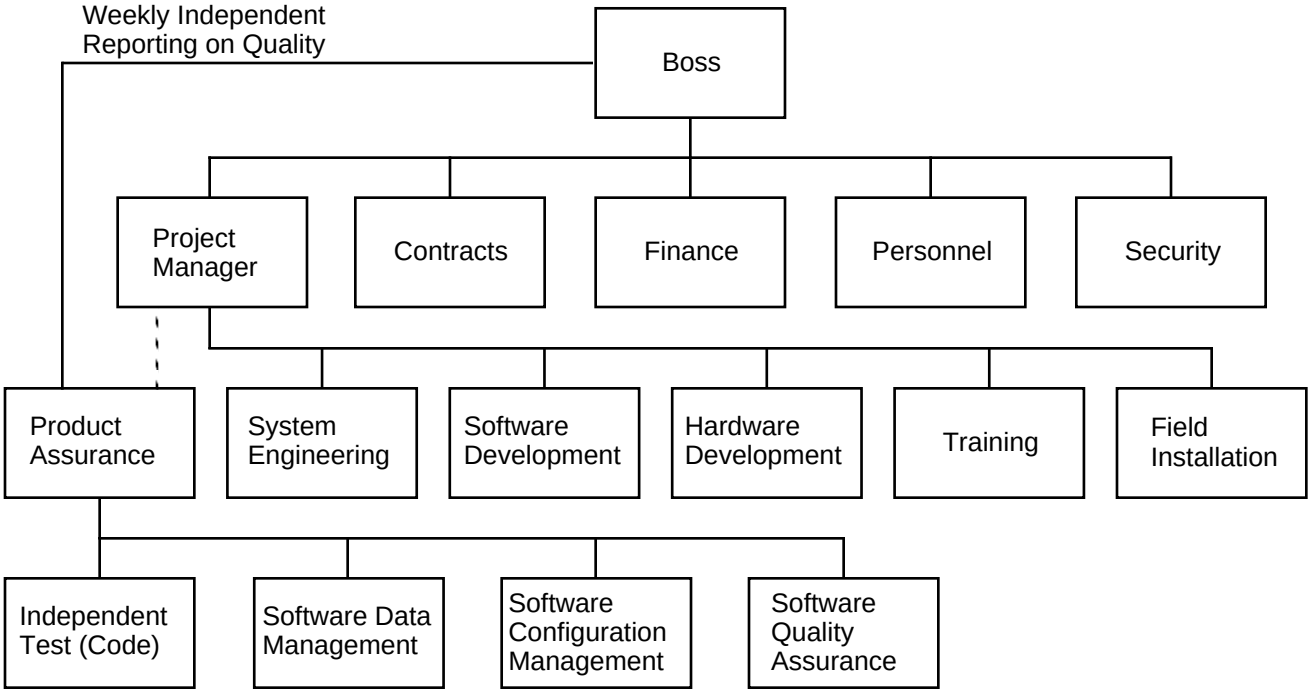


Figure 3-1. Example of an Independent Test Organization

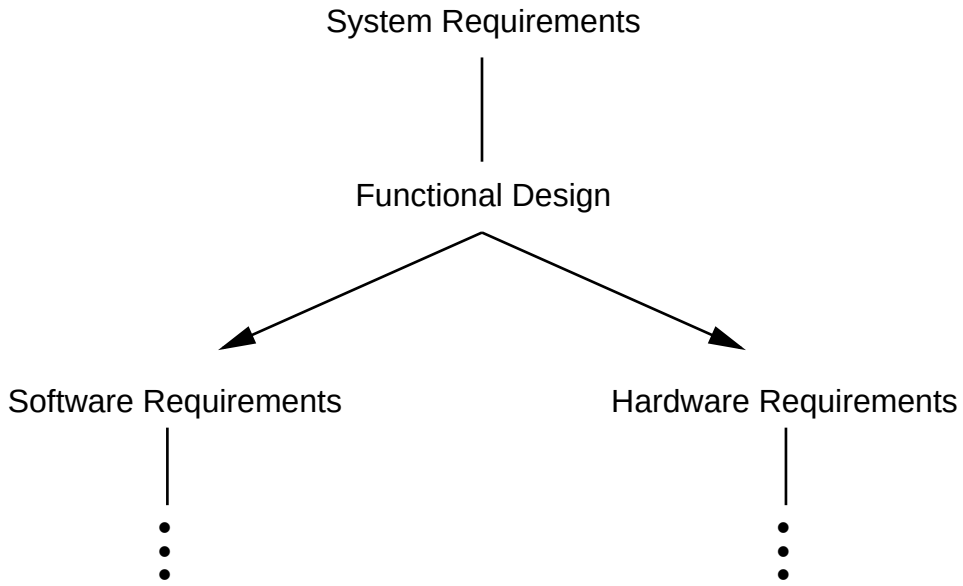


Figure 3-2. Level R Testing

3.3.2. Level 0 Testing

Level 0 testing consists of peer reviews, design walkthroughs, code walkthroughs, and documentation walkthroughs. This procedure is informal for most systems, but formal walkthroughs are required for trusted systems.[2] The purpose of walkthroughs is:

- To validate that the design meets system requirements.
- To verify design logic.
- To verify interface design.
- To verify adherence to design standards and documentation standards.

Rigorous inspections are used and documented by means of a technical conference report. The following types of errors are emphasized: missing requirements, unsatisfactory requirements, logic errors, database design errors, interface design errors, and standard adherence errors.

Level 0 design walkthroughs occur for units (consisting of about 100 lines of code) and components (consisting of about 10 units). Attendance varies according to the complexity and criticality of the object under consideration. The minimum is the software engineer and a quality assurance person (tester). Additional personnel who may need to be present are the software development manager, a customer representative, the software configuration manager, maintenance personnel, and any required invited experts.

The benefits of level 0 testing of the software design are to:

- Find design and interface errors in the development process.
- Reduce coding errors by a factor of ten.
- Ensure that standards are adhered to.
- Provide the tester with enough information to prepare unit test descriptions.
- Reduce testing time and cost.
- Verify that software requirements are satisfied by the design.
- Identify and eliminate poor techniques that may adversely affect maintenance.
- Share techniques and ideas among programmers.
- Promote team member interaction through the walkthrough process.

Studies by IBM and IEEE have shown that it is 10–30 times less expensive to correct design errors during the design walkthroughs (before coding) than after coding is completed.

Level 0 code walkthroughs include the same basic set of skills, though emphasis shifts from process and systems knowledge to software knowledge. Code walkthroughs review the same items as was done in the design walkthrough—plus the code itself. The walkthrough is held after the software engineer affirms that the code is completely "debugged." Benefits include:

- Self-esteem of the programmer becomes tied to defect-free code. As a result, quality is built-in.
- Up to 85% of the implementation errors (coding defects) can be identified and corrected before formal testing begins. Errors found at this level are inexpensive to correct.
- The implementation of the design can be validated.
- Unit testing becomes possible within time and schedule constraints.
- The code-control and the testing process begin with the results of the code walkthrough.

If the walkthrough process can be automated (partially or fully), the resulting data provides the tools to perform effective maintenance.

SED believes that the entire review and testing process is necessary in order to produce defect-free code.

[2] CSC uses the term "trusted system" to describe a system with very high safety or security requirements.

3.3.3. Level 1 Testing

Level 1 testing is carried out at the unit level. SED uses the DoD-STD-2167A definition of unit: the "smallest logical entity specified in detailed design which completely describes a single function in sufficient detail to allow implementing code to be produced and tested independently of other units." Units average 100 executable lines of code (ELOC), and generally should not exceed 200 ELOC.

The purpose of unit testing is as follows:

- To check for mathematical and logical correctness.
- To validate performance against system and subsystem requirements.
- To measure memory usage and throughput rates.
- To exercise every program path.

Testing tools are used to generate the list of program paths, and to generate the values of variables that must be used to force the program to execute the different paths. The following types of errors are emphasized: mathematical, requirement implementation, parameter passing, and path errors. Test execution includes path analysis, boundary class testing (stress testing), and parameter testing.

Benefits of level 1 testing are as follows:

- Identify logic errors.
- Support maintenance by using the data produced by the testing process.
- Resolve man–machine interface questions.
- Build "engineering" confidence into the code.

Level 1 testing is believed to cut system integration costs and schedule in half. It can remove 75% of the coding errors and 20% of the design errors.

3.3.4. Level 2 Testing

Level 2 testing—sometimes called "integration testing"—is carried out at the component level. A component is a collection of approximately ten related units.[3] Two types of component testing are used: testing each component as a collection of units, and testing the interfaces between components.

The purpose of component testing is as follows:

- To check interfaces between units.
- To check interfaces between components.
- To check global variable usage.
- To check flow of control.
- To verify applicable system requirements.
- To verify parameter passing between units.
- To verify parameter passing between components.

Requirements tracking tools are used to generate test cases for components and component interfaces.

The benefits of level 2 testing are as follows:

- Identify and correct interface problems.
- Identify and correct global variable problems.

It is easy to analyze system throughput and timing problems at this stage of testing. The test activity generates data (test procedures and test cases) that are needed later during maintenance. SED uses an automated requirements tracking tool, which is used to generate test cases to test interfaces between requirement matrix and the code. The results of the test activity provide data which will be used later for stress testing.

3.3.5. Level 3 Testing

Level 3 testing consists of program and subsystem testing. (SED uses the MIL-STD term "configuration item.") A program or subsystem is defined to be a stand-alone portion of the system which is "capable of performing specific

---

[3] CSC uses the following definition of component, taken from DOD-STD 2167A: "A functional or logically distinct part of a computer software configuration item. Computer software components may be top-level or lower-level." A software configuration item is software "which is designated by the contracting agency for configuration management."

operational tasks in support of an identifiable mission objective."

The purpose of subsystem testing is:

- To allow the customer to "buy-off" on the software subsystem.
- To verify the correct implementation of every software requirement.
- To demonstrate the capabilities of the software subsystem.
- To verify that the throughput and timing requirements are met.
- To verify the correct interface to the system environment.

Level 3 testing is frequently carried out with the assistance of a simulator. The system under test consists of both hardware and software in a semi-operational setting. Sensor and actuator lines are connected to a simulator instead of "real" devices. The simulator generates sensor signals, accepts actuator signals, and evaluates the correctness and timeliness of the hardware/software system being tested. See Figure 3-3.

The benefits of level 3 testing are as follows:

- Verify the execution of the software subsystem in a simulated environment.
- Verify the execution of the software subsystem in a stressed environment.
- Verify the correct implementation of all the software requirements.
- Verify the interfaces to other subsystems.

The availability of test information allows the system engineers to do fast hardware checkout. The test products provide a demonstration system for the customer, provide a "what-if" capability, and can provide a potential field (user) trainer. The test cases and test data provide needed information for the maintenance testing activity.

3.3.6. Level 4 Testing

Level 4 testing is carried out on the complete system, using a real or simulated environment. The purpose is as follows:

- To permit customer "buy-off" on the complete system.

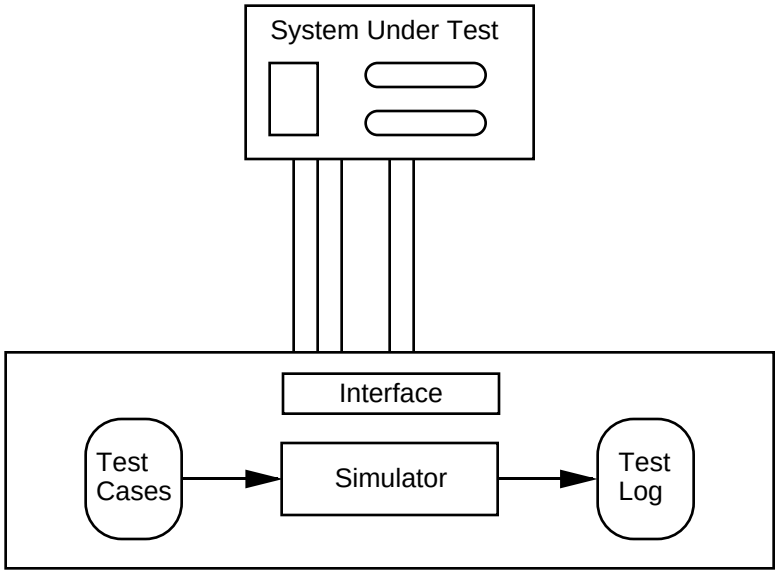


Figure 3-3. Example of a Level 3 Test Design

- To verify the correct implementation of the system requirements.
- To demonstrate system capabilities.
- To verify the hardware/software interfaces.
- To verify that the system works in both a normal and stressed environment.

The benefits of level 4 testing are as follows:

- Verify the operation of the system in a real environment.
- Verify the man–machine interface.
- Verify that the system correctly implements all system requirements.
- Verify the interfaces to hardware subsystems.

The system test cases provide a training scenario for the user community. The result provides a demonstration system for the customer, and provides needed data and configuration for future maintenance testing.

3.4. Summary of Testing Costs

SED testing philosophy can be summarized as follows. All levels of testing are required to obtain a defect-free software product.

- Every requirement must be implemented, verified, and validated.
- This testing method shortens system integration time.
- The test cases and test data that result from the different levels of testing provide useful maintenance tools and procedures for the customer.

The amount of testing discussed in Section 3.3 will be about one-third of the total software development costs, assuming the tools are already developed. Their experience and data show that the benefit is to reduce system life cycle costs by an average of 55%.

Each level of testing requires approximately one-half the effort of the previous level. This can be expressed as the following equations, where C_i is the estimated effort of testing at level *i*:

$$C_1 = \tfrac{1}{2} C_0$$
$$C_2 = \tfrac{1}{2} C_1 = \tfrac{1}{4} C_0$$
$$C_3 = \tfrac{1}{2} C_2 = \tfrac{1}{8} C_0$$
$$C_4 = \tfrac{1}{2} C_3 = \tfrac{1}{16} C_0$$

The table in Figure 3-4 provides estimates of the percent of different error classes found by the different levels of testing.

Testing Level	Error Classes						
	Total System Errors	Design Errors	Coding Errors	MMI Errors	Interface Errors	Database Errors	Performance Errors
0	50–70%	~70%	~10%	~15%	~10%	~10%	~5%
1	70–80%	~20%	~75%	~50%	~10%	~50%	~5%
2	80–90%	~5%	~10%	~15%	~50%	~30%	~20%
3	90–95%	~4%	~2%	~15%	~20%	~5%	~50%
4	95–99%		~2%	~4%	~9%	~4%	~19%

Figure 3-4. Percentage of Errors Found at Different Levels of Testing

An illustration of software costs incurred using the testing methods described here was given by SED at the Fifth International Conference on Testing Computer Software in June, 1988. This project consisted of 184,000 lines of code. The effort figures given next exclude project management, project control, system support, licenses, and procurement.

The development effort consisted of approximately twelve people:

Development Team:	5–7 programmer/analysts
Test Team:	3–5 programmer/analysts
QA/CM:	3 people

Costs were reported in various ways. First, cost by function was distributed as follows:

Development:	48%
Testing:	30%
QA/CM:	22%

Next, the development costs were reported by development phase:

Requirements:	14%
Preliminary Design:	21%
Detailed Design:	36%
Coding:	29%

Finally, the test costs were reported by test activity:

Requirements Analysis:	10%
Test Documentation:	10%
Review Development Documentation:	12%
Unit Test:	26%
Integration Test:	6%
Testing Tools:	36%

To summarize, the keys to successful testing are the following:

- Use highly qualified people.
- Focus testing on requirements.
- Integrate testing with development.
- Monitor schedule and budget. Maintain flexibility and the ability to adapt to change.
- Automate the testing activity.

Some problem areas that SED has found are:

- Living within the original cost and schedule estimates for testing.
- Establishing a level of testing which is reasonable and sufficient for the applications.
- Making estimates more detailed than available data justifies.
- Convincing both the customer and company management that extensive testing is beneficial.

3.5. Configuration Management[4]

Product assurance includes the disciplines of quality assurance (QA), configuration management (CM), data management (DM), and independent test. The purpose of product assurance is to "support the development team by assuring that a quality product is being produced, controlled, and delivered." SED defines CM as "the means through which the integrity and continuity of the design, engineering and cost trade-off decisions made between technical performance, producibility, operability, and supportability are recorded, communicated, and controlled by program and functional managers." More specifically, CM is a discipline that applies both technical and administrative direction and surveillance to three areas:

- Identify and document the functional and physical characteristics of a configuration item.
- Control changes to those characteristics.
- Record and report change processing and implementation status.

[4] The information in this section is taken from a CSC presentation to LLNL on December 6, 1989.

Configuration management tasks can be divided into five areas, as follows:

- Configuration identification. List all configuration items—hardware and software. Create numbering schemes.
- Configuration audits and reviews, which includes walkthroughs, test audits, functional configuration audits, physical configuration audits, and responsibility matrices.
- Metric collection and reporting. This will need to be tailored to the project requirements. Examples of metrics include estimated lines of code, number of pages in documents, and number and type of errors.
- Configuration status accounting. Prepare and deliver reports on the configuration effort.
- Configuration control, which includes baselining, code and document control, and release control.

Configuration management activities are carried out at each phase of the software life cycle—for SED, a standard waterfall life cycle. The following tasks are recommended by SED for each life cycle phase.

Requirements Phase

- Write software CM plan (SCMP).
- Set up reference libraries and control libraries.
- Establish configuration identification scheme.
- Identify CM tools and CM forms.
- Identify metrics requirements.
- Benefits of CM in requirements phase:
  - Controls are planned, staffed, and operating.
  - Project memory is established.
  - Corporate memory is established.
  - Metrics are collected.
  - A sanity check is established.

Preliminary Design Phase

- Set up status reporting.
- Establish code libraries.
- Write software delivery documents.
- Implement SCMP.
- Collect design metrics.
- Control requirements (e.g., tracking matrix).

Detailed Design Phase

- Set up version description documents.
- Implement CM tools.
- Operate libraries.
- Benefits of CM in design phase.
  - All requirements are documented.
  - All requirements are under control.
  - Productivity is improved because of the team's focus on approved requirements.
  - Requirements growth is controlled; as a result, budget and schedule are controlled.

Implementation and Testing Phase

- Conduct walkthroughs.
- Collect metrics and history files.
- Manage run-time libraries.
- Prepare for and conduct functional configuration audits.
- Prepare for and conduct physical configuration audits.
- Collect failure metrics.
- Benefits of CM in implementation and test phases.
  - Facilitates design and code walkthroughs.
  - Provides a more effective test program.
  - Tracks all discrepancies and makes sure that corrections are made.
  - Ensures tracking of all action items.

4. INTERNATIONAL BUSINESS MACHINES FEDERAL SYSTEMS COMPANY

4.1. Introduction

The IBM Federal Systems Company (FSC) has been working on process and quality improvement since its beginnings in the early 1960s, and is still working on these today. The process never ends. Actual data on the effects of process improvement exist beginning in the mid 1970s.

The process of improvement over the past two decades has gone through a number of phases, each adding a new element to the overall development process. About 1976, FSC began working on development enhancement, followed a few years later by adding an element to enhance requirements analysis. In the early 1980s, independent verification and quality management enhancements were added. By 1984, the focus had expanded to include the entire development process.

FSC has found that the effect of process change may take up to two years for results to appear in the delivered products. This is significant, since it implies the need for long-term management commitment to improvement, constant monitoring and effort, and considerable patience. It also implies that changes cannot be introduced too rapidly or the effect of an individual change cannot be measured and evaluated due to the possible effects of other changes.

FSC uses a formal improvement model, consisting of four major steps:

1. Assess the present—where are we today?
2. Model the future—where do we want to go?
3. Plan the transition—how do we get there?
4. Integrate the change—make it so!

FSC has a goal of error-free software, defining quality as "conformance to customer requirements." Other definitions of this term, such as "conformance to customer expectations," exist. Information presented by FSC must be understood in the context of the former definition, and no other. The customer (NASA) sets the goals for reliability, availability, maintainability, and performance. If these goals are met, then the product is considered to be of high quality.

An important aspect of the FSC process is that of detection and removal of errors in the software documents and code. FSC divides errors into three sets:

1. Early detection—errors detected before a new system build occurs.
2. Process errors—errors detected between a new system build and delivery to the customer.
3. Product errors—errors detected after delivery to the customer.

The effect of FSC's process improvement over several decades is that early detection errors now run about four per thousand source lines of code (4/KSLOC), process errors are about 1/KSLOC, and product errors are nearly nonexistent.

The improvement in error rates is shown in Figure 4-1. This chart needs to be read carefully. There is about one product release per year, and errors are counted against that release as long as they can be traced back to it. This means, for example, that a version released in 1985 has been counting product errors for 8–9 years, while a version released in 1993 has barely begun counting product errors. The result is that the product error curve may be biased somewhat. However, shuttle systems within the past five years have shown virtually no errors discovered after first flight usage. The number of early detection errors in each release and the number of process errors in each release do not have this bias since, by definition, these error counts do not change after release.

4.2. Key Measures to Improve Product Value

It is not possible to manage software reliability effectively until the development organization has demonstrated its capability to consistently carry out four actions. The organization must consistently produce products that (a) comply with their requirements, (b) are within budget, (c) are on time, and (d) are at an appropriate quality level. The developer must be able to carry out all

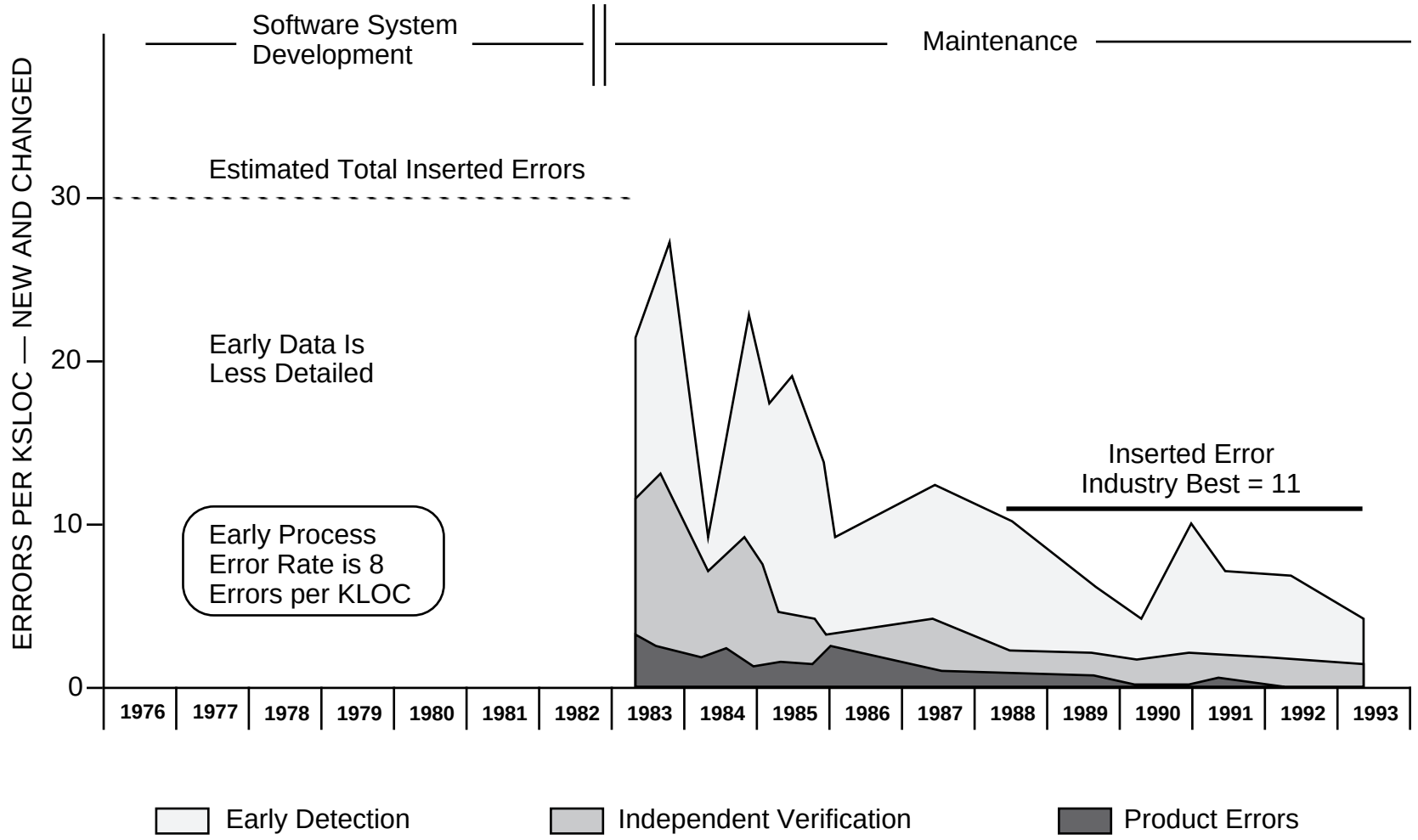


Figure 4-1. On-Board Shuttle Software Error Measurements

of these actions concurrently for every project. Quality improvement is a multi-step process, and each step needs to be completed before going to the next. Six steps for improving quality are listed below, with some key phrases that describe them. FSC disagrees with the oft-quoted statement "quality is job one." It is actually about job six, and jobs one through five must be understood, controlled, and managed before it becomes possible to manage quality.

Initiation

- Define the processes and the process models.
- Stabilize the process.
- Standardize.

Measurement

- Define measures consistently.
- Relate measures to key goals.
- Relate measures to processes used.
- Measure with integrity.
- Retain the measurements in usable form—keep a corporate history.

Modeling

- Relate the data models to the process models.
- Relate both data models and process models to project parameters.
  - Function size.
  - Complexity.
  - Criticality.
  - Process proficiency.
- Relate the models to project schedules.

Prediction

- Calibrate models to new problems.
- Tailor the process to new problems.
- Forecast the future, based on history and product definition.
- Establish a plan, based on the process model, for achieving the forecasts.

Control

- Measure.
- Evaluate.
- Take action.
- Manage change.

Improvement

- Predict the impact of improvements.
- Redo the first five steps after the improvement has been implemented.
- Isolate improvements to one element at a time.

4.3. Software Process Improvement

IBM/FSC has devoted a great amount of time, thought, and work to process improvement over the last fifteen years. Considerable data exists on the onboard shuttle (OBS) software. By 1976, when the data begins, FSC had a sound foundation for process improvement in place.

Figure 4-2 gives a summary of process improvements, showing when each was added to the FSC process and to what it related. This is discussed in Sections 4.3.1–4.3.4, which contain lists of "lessons learned" relating to development process improvement, requirements management, independent verification, and quality management.

4.3.1. Development Process Lessons Learned

- Inspections were key to the improved product quality for OBS.
  - The inspection process must be defined and enforced.
  - Management must support and enforce the focus on quality, meeting requirements, and finding errors early. The reward system must agree with this philosophy.

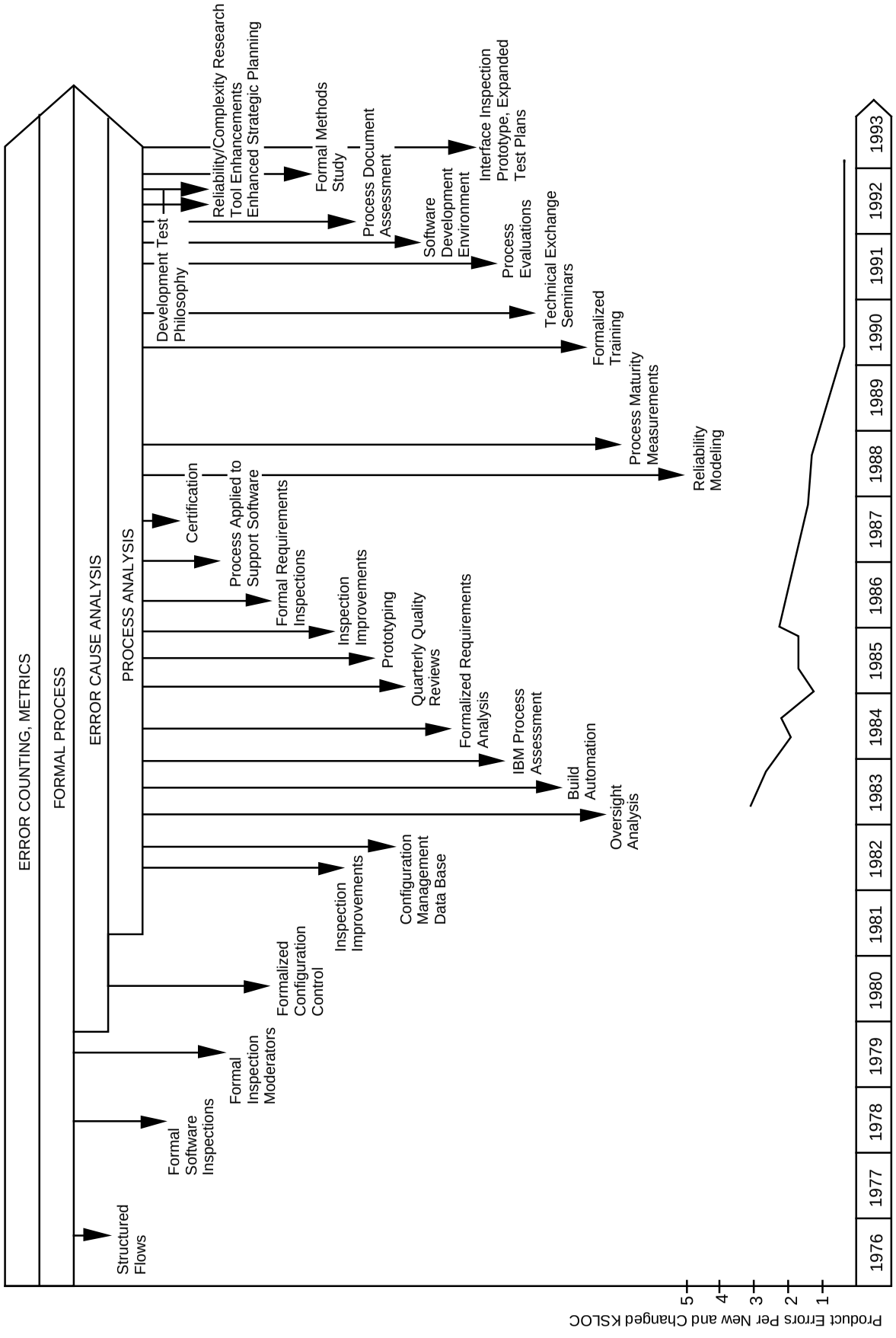


Figure 4-2. Introduction of Process Improvements

  - Employees that perform the process are very good at improving the process.
  - Data collection must be balanced between resources required to amass the data and the use of that data.
  - Inspections are tailored in different functional areas to address features that are unique to the functional area. This enhances productivity and focuses the human resources where they are most needed.
  - Automation of configuration control and error checking whenever feasible also frees human resources for other tasks.

- Configuration management
  - OBS was able to institute CM very early, using manual procedures. The understanding that CM was absolutely necessary was key to the product quality.
  - The CM concepts are basic to software process control and are fundamental to the continued success of OBS. Quality enhancements rely, in almost all cases, on some data from the CM database.

- Process Considerations
  - Processes in less mature organizations must be more controlled and perhaps less flexible than those in mature organizations. One must first bring a process under control before one can manage that process.
  - Significant results are achieved when a mature process is transferred to other projects.
  - Mature processes continue to change with new situations. Documentation, process owners, and management must change as necessary to continue productivity and quality improvement.

4.3.2. Requirements Management Lessons Learned

- Improving the requirements analysis (RA) process also improves downstream processes.
- Having RA as part of design and code inspection teams provides another opportunity to identify requirements issues before the code is delivered.
- Having the people doing RA also perform final testing gives a unique perspective to see if the software will meet the user's needs.
- Continuous flow of process improvement ideas has been fostered by the process team and by focusing on documenting issues and suggestions.
- FSC still needs a consistent methodology for documenting requirements across projects and organizations.
- Requirements inspections decrease requirements errors, improving product quality and eliminating rework.

4.3.3. Independent Verification Lessons Learned

- A structured V&V process leads to producing higher-quality software. The test plan should be developed early in the life cycle; formal test cases must be developed, executed, and analyzed; V&V results should be peer-reviewed; there should be a final review by the customer; and records should be retained for future use.
- It is necessary to maintain independence between process functions. That is, the requirements analyst, the developer, and the verifier must have organizational independence from one another.

4.3.4. Quality Management Lessons Learned

- The development process must be under control before process enhancements can be managed effectively. The following are necessary to have control over the software process:
  - A formal configuration control process must be implemented.

  - A configuration management database is essential for productivity and automation.
  - Measurements and error counting must begin as the project begins.

- An important aspect of process improvement is defect cause analysis. The ability to learn from errors is key to effective improvement.

- External review of the process is valuable. It brings insight that may not be available internally.

- There is no substitute for management attention and focus on the quality of the product and the needs of the customer. Employees do what management rewards them for.

- A mature organization can always find other processes that have information and techniques that can benefit and improve even very good processes.

- Automation can facilitate process improvement. Process enactment, that is, automating the process in the environment used to develop code, will continue to improve quality and productivity. It frees the human mind to do the tasks that are either too complex for automation or involve judgments that are not easily parameterized.

The following summarizes FSC lessons learned from metrics analysis:

- Cost and quality can be related, allowing realistic costing of desired quality levels.

- Careful interpretation of metrics can yield extremely effective process improvement feedback.

- Trend analysis of metrics can provide valuable insight into areas requiring special study and emphasis, and can aid in management decision making.

- A defect prevention approach can be made much more effective by using metrics feedback to identify unrecognized software failure modes.

- Metrics actually affect workers' views of their own worth, in turn psychologically influencing work quality.

- Testing must be selectively focused as a result of metrics analysis.

- A metrics analysis process can be only as good as the data collected and retained in the metrics databases.

- The most effective productivity initiative is error prevention and early detection.

- Empirical cost and quality predictions are feasible only if proper metrics data are available.

- Frequency of metrics reviews across programs must be optimized for project-specific goals.

- Cycle-time influences can be determined and managed when appropriate metrics are made available for analysis.

4.4. Software Reliability

Flight software failure modes are divided into three severity levels. Severity 1 failures have severe vehicle or crew performance implications; these include loss of vehicle and death of crew members. Severity 2 failures affect the ability to complete mission objectives, but are not safety issues. Severity 3 failures are the remaining failures that are visible to the user, but have minimal effects on the mission. Only severity 1 failures are safety-related.

NASA and FSC log every software discrepancy. (A discrepancy is a suspected error, fault, or failure.) The discrepancies are traced back to the actual software, and are tied to the actual writing of that software. Since FSC software releases consist of multiple modules of code written at different times over periods of years, this trace-back is essential to finding out what went wrong, and whether process enhancements added after the date the error was introduced would have prevented the introduction.

Testing alone cannot prove that software has no errors. Half of all shuttle discrepancies are found by static analysis, not testing. Adding more testing can asymptotically reduce the risk of

software error, but cannot eliminate it. Quality must, therefore, be built into the software.

Before the software can be used on a shuttle mission, FSC is required to certify, to the best of its knowledge (based on application of extensive analysis, testing, and prediction techniques) that no severity 1 errors are present in the software.

Reliability prediction is done through modeling. FSC has found that the Schneidewind model best fits their data. The models predict that at most one mission per thousand will encounter a severity 1 failure. However, this prediction is based on testing results where the testing is mostly of off-normal events. A more accurate interpretation of the model appears to be that at most one mission per thousand *that contain off-normal events* will encounter a severity 1 failure.

Quality must be built into the software by using appropriate software engineering methods. There is no single technique that has led to FSC's success; rather, success has been a result of doing all of the techniques, and doing them well. Techniques critical to high reliability are:

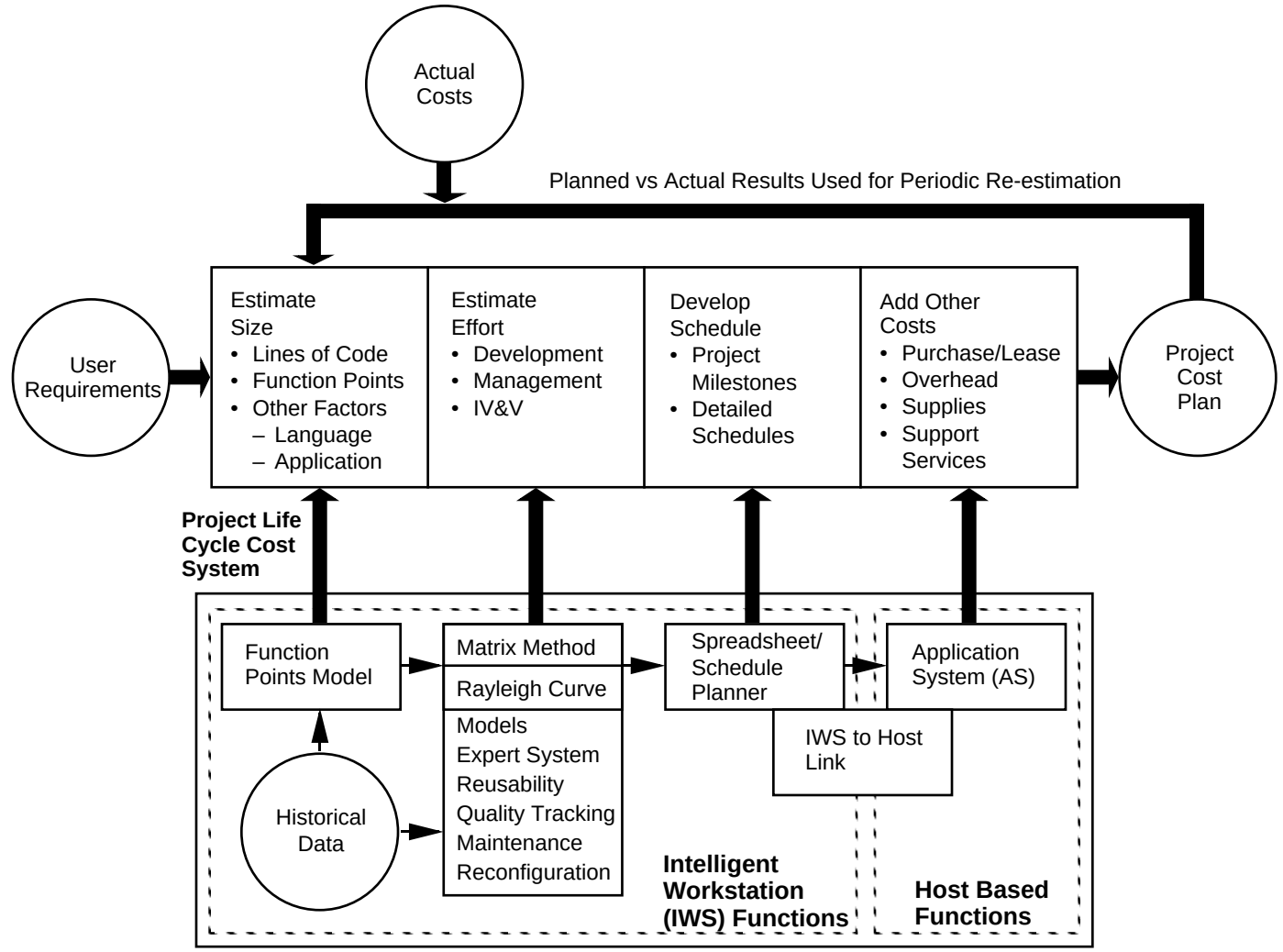
- Structured software development process.
- Rigorous configuration management.
- Failure mode identification, analysis, and elimination.
- Oversight discrepancy analysis and feedback for process correction.
- Automation of software production processes.
- Independent verification and validation.
- Quality-monitoring metrics and interpretation.
- Inserting new technology to upgrade the development process.
- Management commitment to an "error-free" culture.
- Employee education and rewards.
- Focus on the development process—especially the "front end."
- Continual measurement and analysis of the process.
- Elimination of defect causes.
- Use of industry standards and competition to assess quality.

4.5. Aspects of Software Development

4.5.1. Cost and Quality Planning

FSC has developed and refined a method for predicting and controlling development costs and schedule. If these are not controlled, the project will almost certainly exceed budgets, at which time testing is generally aborted and quality suffers. The general approach is shown in Figure 4-3. The following list provides the elements of FSC's costing method:

- Decompose requirements into functional areas.
- Relate functional areas to known functional areas where the development organization has experience.
- Estimate function size in terms of source lines of code (SLOC).
- Assign complexity and criticality levels to the different design and code elements.
- Assign functions to different product releases.
- Select the project productivity factor based on complexity, release, use of COTS software, and any reused software modules.
- Select a verification factor based on criticality.
- Select other indirect cost factors based on criticality.
- Divide SLOC by the productivity factor to arrive at the direct development labor months required for the project.
- Multiply the development labor months by the verification factor to arrive at the direct verification labor months.



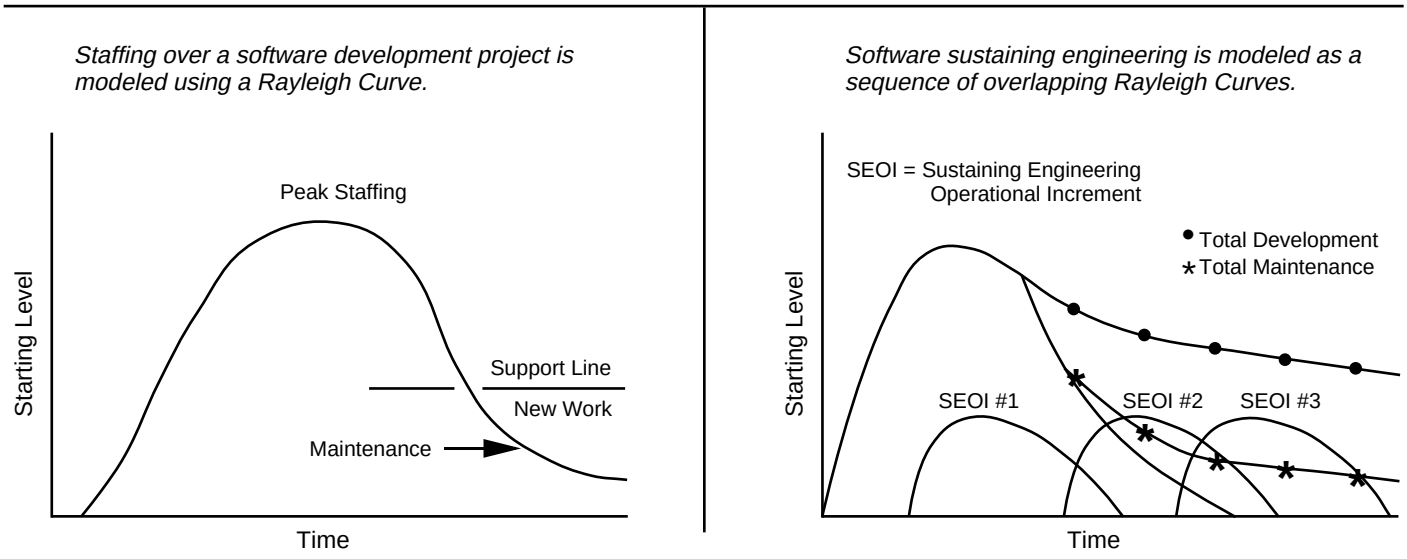


Figure 4-3. Software Life Cycle Costing Methodology

- Add the development and verification labor months together, and multiply the result by the indirect factor to arrive at indirect labor months.
- Add development, verification, and indirect labor months to arrive at the total development labor months. This excludes maintenance and operations activities.
- Use a Rayleigh curve to estimate the project schedule and labor loading.

The amount of effort which must be devoted specifically to quality management can be estimated from the criticality of the project, measured by the required product error rate in terms of errors per thousand source lines of code (KSLOC). For early development phase of large software systems, Figure 4-4 shows the percentage of a project that must be devoted to quality in order to achieve a specified error rate. "Ordinary" products can be produced by devoting approximately 10% of the development cost to quality management, resulting in about one error per KSLOC in the delivered code. Reducing error rates below this requires much larger investments in money, time and labor. To achieve a delivered error rate of 0.1/KSLOC, approximately 60% of the project budget must be spent on quality management. Corresponding curves with lower error rates can be developed for more mature processes, methods, and system maintenance.

There are two types of technical cost, driven by different factors. One is *functional* cost, driven by the size and complexity of the application functions. The other is *schedule* cost, driven by time. These should be separated and managed separately to be successful at managing the development process. *Management* cost is in addition to these technical costs.

4.5.2. Software Requirements Process

Requirements analysis is defined by FSC to be the process of "decomposing, analyzing, and understanding the needs of the customer and exactly specifying them to be correct, implementable and testable." Requirements are a primary driver for many aspects of software design and development, including cost, schedule, skills required for implementation, resources required, the hardware/software architecture which will be used, test plans and schedules, and operational procedures. There are many benefits to requirements analysis. All parties to the development (customers, users, developers) are more likely to share an understanding of the requirements, and to be satisfied with the delivered product. The probability of errors in the requirements is reduced, and the software system is more likely to be completed within budget and schedule constraints.

Errors made during requirements specification can be very costly if not found until later in the life cycle. Figure 4-5 provides a range of expected relative costs, depending on when the error is discovered. Units may be time, dollars, or another measure of cost. FSC uses a six-step process for requirements analysis. This is shown in Figure 4-6, and discussed below.

Requirements conception will result in an understanding of the operational need for the software system. Architectural options for the entire system (hardware, software, people) are examined, and the preferred architecture is determined. An outline of the intended software system is sketched.

Requirements generation produces the actual requirements specification as specific requirements are determined and documented. The requirements specification serves as (1) the primary technical input to the design team, (2) the primary input to the software test team, (3) a contractual agreement between the customers and the developers, and (4) the basis for communication among all the parties concerned. Developing the specification is an iterative process, and can require a considerable amount of work. The benefits of a systematic requirements specification are to save resources in subsequent development phases and achieve a final product which satisfies the customer's real needs.

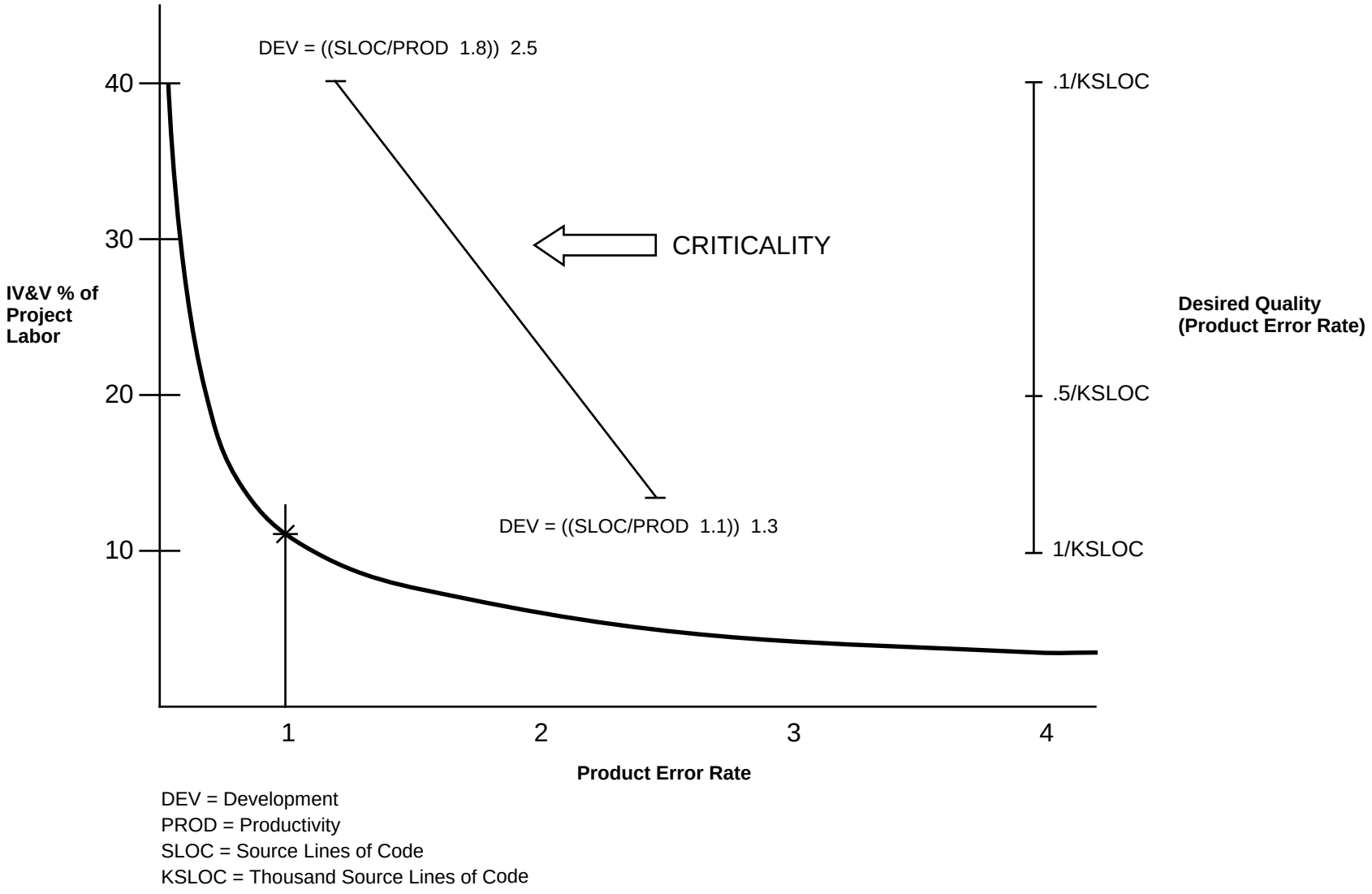


Figure 4-4. Software Criticality vs. Cost for Early Development Phase of Large Software Systems

Life Cycle Phase in Which Error is Detected	Relative Cost to Correct Error
Requirements	1
Design	3–6
Coding	10
Development Testing	15–40
Acceptance Testing	30–70
Operations	40–1000

Figure 4-5. Cost of an Assumption Error in the Requirements Phase

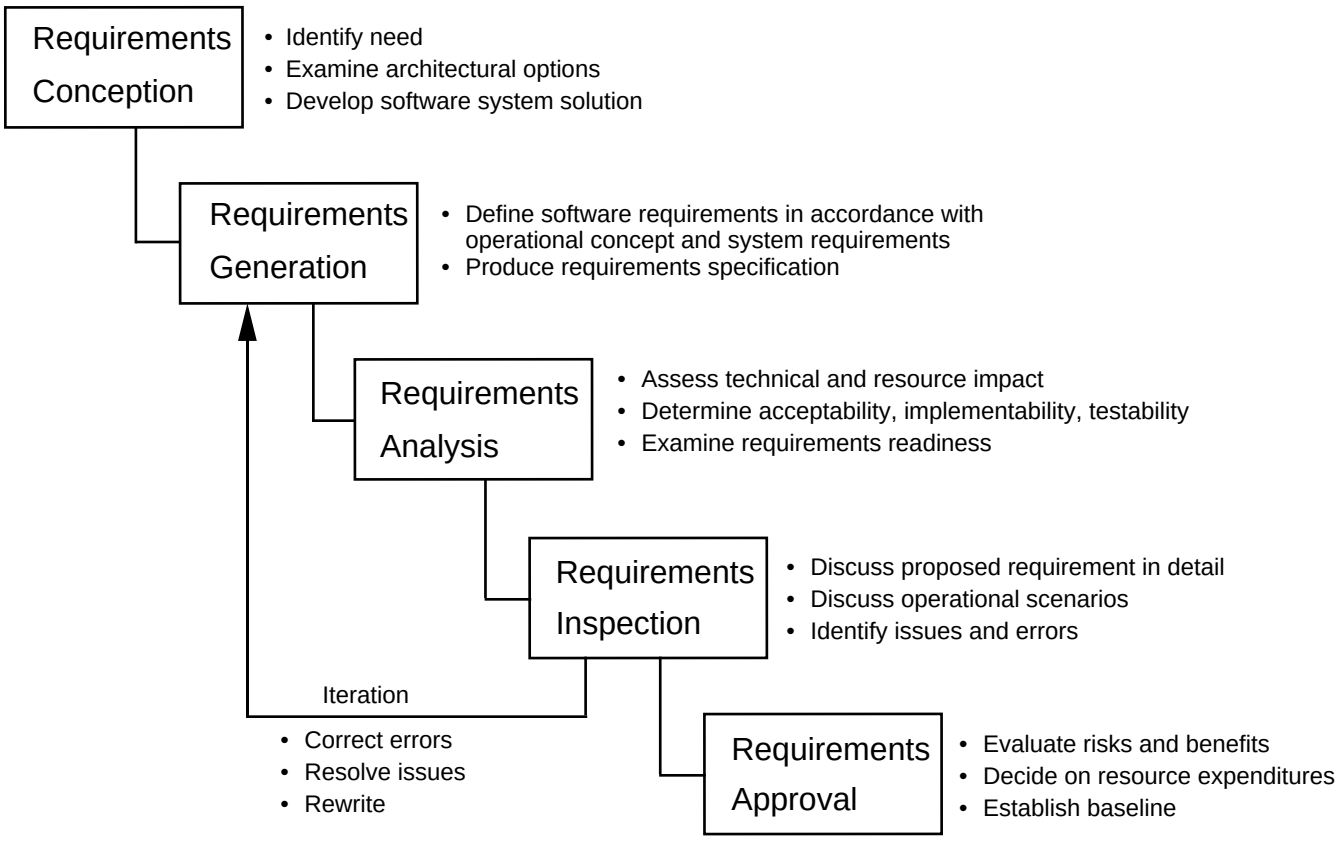


Figure 4-6. Requirements Analysis Process Steps

The following list of key words describes a good software requirements specification:

Unambiguous	Traceable
Consistent	Concise
Complete	Verifiable
Correct	Logically closed
Understandable to customer	Design independent
Organized	Non-redundant
Modifiable	Justified
Traced to its origins	

Requirements analysis is a detailed technical examination of the requirements to assess the feasibility of implementing them. Inputs to the analysis include cost and resource assessments, risk assessments, and a readiness assessment—is the requirement ready for implementation? Many methods are available. A functional decomposition approach gradually decomposes high-level functions into lower-level functions by increasing the amount of detail. Object-oriented analysis creates a data model based on objects defined in the application domain. An event–response analysis indicates how the system will respond to all foreseeable stimuli. A data-driven analysis concentrates on the definition of data structures and the actions that take place on the data. In most cases, more than one type of analysis will be required.

Requirements inspections should be carried out from time to time during the generation and analysis activities. The intents are to determine if (1) changes are written against the most recent requirements base; (2) all necessary requirements are included; (3) the requirements meet the intention of the customer (author); (4) the requirements consider all related options and scenarios; (5) the requirements are technically clean, understandable, and maintainable; and (6) the requirements are consistent across all areas. Inspections can be done by a formal page-by-page specification review, prototyping, customer walk-throughs, requirements simulation, structured requirements analysis, formal methods (mathematical proofs), or combinations of these.

There are four primary sources of requirements errors. These are (1) failing to capture all of the requirements (particularly reliability, maintainability, portability, efficiency, and user-friendliness requirements); including implementation details; working in isolation from the customers; and over-specifying requirements. Industry data quantifies the types of non-clerical requirements errors:

Incorrect fact	49% of errors
Omission	31%
Inconsistency	13%
Ambiguity	5%
Misplaced requirement	2%

Requirements are rarely written correctly the first time, so an iteration step should be included. The process of generation–analysis–inspection–iteration should continue until no more errors are found. At this point, a formal approval process should take place to evaluate the risks and benefits of proceeding with software development, determine what resources will be required for development, and decide whether or not to proceed.

4.5.3. Error Cause Analysis and Defect Prevention

One goal of the FSC development process is to produce error-free software. An important aspect of this process is to use information about software errors to improve the development process. The existence of an error in the delivered product can be viewed as a failure of the development process. Why was the error inserted into the product? Why was it not caught during development? How can the process be changed to prevent the error from re-occurring?

Defect prevention actions require a disciplined, repeatable process if they are to be effective. Understanding how an error was inserted into the product requires thorough documentation; tracking of process, product and defects; and analysis of the root cause of the error instead of the symptom.

FSC collects a considerable amount of information about each defect that is discovered. Some of this is listed here:

- Description of the defect
  - What was the problem?
  - When was the problem found?
  - Who found it?
  - Where was it found?
  - How was it found?
- Analysis of the defect
- Information about the defect
  - What was the potential severity of the defect?
  - What type of defect occurred?
- History of the defect
  - During which activity was the defect found?
  - During which activity was the defect introduced into the product?
  - During which activity should the defect have been found?

Over time, patterns and trends of defects can be detected, and corrective action can be planned, proposed, and carried out.

There is no magic to defect prevention. It requires a continuing effort over many years. Each root cause of inserting defects into the software product must be found individually and eliminated individually. Work habits must change, and the development process must change. It typically takes about two years for process changes to become fully incorporated into everyday practice, provided that the corporate culture encourages such change and management is both understanding and committed. However, when viewed over a period of 10–15 years, the cumulative effect of the improvements at FSC is quite impressive.

4.5.4. Obstacles to Highly Reliable Software

FSC was asked "What are the most important obstacles to producing highly reliable software?" Four obstacles were mentioned:

- Poor software project management.
- Project driven by schedule rather than quality requirements.
- Failure to control the establishment and content of the software product baseline.
- Failure to track software errors and error causes. Failure to learn from these mistakes.

5. TRW

5.1. Introduction

TRW is a corporation of 15,000–20,000 people organized into five divisions. The primary customer is the Department of Defense (DoD), so DoD standards and contracting requirements govern many of TRW's software development methods. Each division is responsible for its own software development; a corporate-wide activity attempts to coordinate the effort and promote standardized development methods.

5.1.1. Software Engineering Process Groups

Process improvement goals are developed through the software engineering process groups (SEPGs) attached to each division, and through the individual development projects. The SEPGs tend to work on broad process issues, while the projects tend to deal only with the particular project domains. A corporate-wide SEPG coordinates the activities of the divisional SEPGs and attempts to reduce duplication of effort.

The divisional SEPGs regularly evaluate the engineering process and identify potential areas of improvement. They create and direct task teams to study and implement process improvements. They evaluate new technologies, tools, techniques, and methodologies for possible use on TRW projects. Some examples of process improvement goals are:

- Standardize software engineering policies and practices.
- Develop a consistent reuse program.
- Develop a consistent training program.
- Develop a better software engineering environment.

- Develop a consistent approach to software cost and size estimation.

- Develop a consistent metrics program, including process instrumentation and higher-level metrics.

Progress in a project is measured through the completion of low-level milestones. Success can be measured through observed improvements in productivity, perceived customer satisfaction, latent errors, time to correct errors, and other parameters.

TRW is committed to the Total Quality Management approach to improving the software development process. Currently, the SEPGs are engaged in process improvement and technology insertion efforts. They are striving to advance TRW's software maturity, and monitoring cost implications in competitive procurement contracting. TRW appears to be undergoing a shift from custom development to purchase and integration, and the SEPGs are monitoring this transition. That is, rather than write new software, TRW expects to buy existing commercial off-the-shelf (COTS) packages and integrate them into a system that addresses the customer's requirements. There is a great deal of uncertainty about using COTS software in safety-critical applications. TRW believes that such use is probably going to occur, but it will be quite difficult to understand the implications.

5.1.2. Highly Reliable Software

TRW is going through a paradigm shift, which is culturally difficult. The driving factor is cost; in order to compete, TRW needs to cut costs. This means that it must reduce programming—buy COTS software and integrate the packages. TRW considers it important to move into reuse of existing products and software packages. If it is not possible to buy or reuse software for a new application, TRW's experience suggests that existing tools and fourth-generation languages should be used in order to control costs.

Throughout industry, software development efforts have usually been underestimated with respect to difficulty. They are consistently under-funded, generally by a factor of at least two. Managers frequently come from hardware backgrounds, and do not understand the difficulty of creating software.

The most severe software errors generally occur in the non-application code, as application code is generally linear and straightforward. Database management systems and communication systems are much harder to create, and consequently tend to have more errors and more serious errors. Operating systems are very difficult to write. Major errors occur in the areas of operating system control and in interfaces between operating systems and application programs.

To achieve safety, one needs to at least double the estimated cost of creating a non-safety system of comparable size and complexity.[5] TRW believes it must concentrate on (application) system control. This should be machine-generated (automatic programming) through a fourth-generation computer-aided software engineering (CASE) system. People are very bad at writing this kind of code; automatic programming techniques generate many fewer errors. It was emphasized that it is critical to certify the automatic programming code generation techniques and tools.

Object-oriented design (OOD) and programming (OOP) probably should not be used in safety-related code. The OOD/OOP paradigm does allow one to keep costs down when making changes during maintenance, but this should not be the primary objective when safety is the issue. The problem is that it is impossible to predict how the safety-related system will really work. For example, the OOP compiler will insert extra code that permits the executing program to make decisions "on the fly" (during execution) which cannot be predicted. (These comments are aimed more at OOP than OOD.) It might be possible to use OOD techniques, and then implement the design in a more predictable manner using a standard third-generation language such as Pascal, Fortran, C, or a limited subset of Ada. This approach costs more, but the result is more predictable.

People can build software that is more-or-less independent of the hardware. TRW needs to write software this way so that the customer can move it with no coding changes from one hardware platform to another; if this cannot be

[5] This means double the "real" cost, not the underfunded cost mentioned above.

done, it becomes impractical to keep up with hardware generational changes. The software industry is on the verge of achieving this.

5.1.3. Certification of Safety-Critical Software

The Ballistic Missile Defense (BMD) Division of TRW creates software to perform safety-critical functions for nuclear weapons systems. This software performs such functions as guidance, launch control, and encryption/decryption, whose failure could have devastating results. Because of this, a rigorous certification process exists. This process, and its supporting standards, are applicable to other nuclear safety-critical systems such as reactor protection system software, since the problems and possible consequences of failure are similar in scope. Some of the objectives of nuclear weapon surety standards are listed below. Analogies exist between the safety requirements for weapon system and the safety requirements for reactor protection systems.[6]

- Prevent nuclear weapons from involvement in accidents or incidents that produce a nuclear yield.
- Prevent deliberate prearming, arming, launching, firing, or releasing nuclear weapons except upon execution of emergency war orders or when directed by competent authority.
- Prevent inadvertent prearming, arming, launching, firing, or releasing of nuclear weapons in all normal and credible abnormal environments.
- Ensure adequate security of nuclear weapons.

BMD believes that methods exist which can be used to attain an extremely high degree of confidence in the software.

- To assure that the software satisfies its functional and performance requirements, use
  - structured development and formal test methods,
  - strict configuration management,
  - formal proofs of correctness.
- To assure that the software satisfies only its requirements, use
  - rigorous and independent tests, analyses, and evaluations,
  - environmental, personnel, and access controls,
  - protection of the software during development.
- To assure that the software remains intact and uncompromised in operation, use
  - built-in tests such as checksums.

BMD believes that safety-critical systems need to be certified. In the following list, the word "system" includes people, procedures, software, computer hardware, application hardware, and all other system components. The purpose of certification is to ensure that:

- The system performs only as intended. Safety-critical systems cannot have any unintended or unexpected functions. They must behave in fully predictable ways.
- Accidents and improper or unauthorized actions are detected when they occur, and if they occur they are adequately precluded from causing a disaster.
- Maintenance actions do not decertify the system. Maintenance actions can cause changes to the software, which may add faults. If this is possible, the changed software requires recertification.
- Upgrades do not preclude recertification. Major changes to the software (such as a new version) will nearly always require recertification. It is important that the upgrades are of such a nature as to make recertification possible.

Certification is a four-step process:

1. Design certification. This occurs after the requirements review and analysis, and determines what will be developed.

[6] Some of the information is the next several paragraphs is taken from a TRW briefing to the NRC on July 15, 1992.

2. Operational certification. This occurs after final testing and validation, and ensures that the system is ready to operate.

3. Decertification. This occurs when a change request has been received, or a fault has been detected. The system is considered non-operational after decertification.

4. Recertification. This occurs after a system or software upgrade, and after retesting and revalidation. Once the system has been recertified, it must also be operationally recertified before being placed back into use (step 2).

If one starts using COTS software, then it will have to be certified also. This is probably much harder than certifying the application.

5.2. The AWIS Project

The Army WWMCCS Information System (AWIS) is an upgrade to the Army's portion of the DoD World Wide Military Command and Control System (WWMCCS). AWIS is a large distributed information processing system written in Ada, which has been under development by TRW for several years. The projected size is three million lines of Ada code. An independent systems assurance organization is used to evaluate processes and products to ensure that the AWIS development effort meets its requirements, complies with project plans, and adheres to relevant DoD and industry standards. An important aspect of this assurance effort is the definition, collection, analysis, and reporting of various metrics.

TRW uses an iterative discovery process in the metrics program. The purpose is to force communications among system developers, break down intra-corporate barriers, cause key issues to surface so they can be noticed and dealt with, and generally to improve understanding of the development process and products. A focus is on process and product quality improvements. Goals are to create a "constancy of purpose," reduce cost and development cycle time, and reduce the scope and cost of inspections and testing. The metrics program attempts to achieve all this by measuring, managing, and maintaining continuous improvement.

A configuration management (CM) system is used to monitor faults and enhancements. Reports from this system provide status, change metrics, and fault statistics from several perspectives, such as by development phase and by product.

The statistical modeling and estimation of reliability functions (SMERF) product[7] is used to track reliability data during formal testing. It is used to collect data on requirements, change requests, error fixing, and so forth. SMERF can produce a great variety of reports. In particular, it is used to make monthly and quarterly reports to DoD.

The following classes of metrics are captured and reported:

- Requirements and requirements volatility—number of requirements and number of changed requirements.
- Software development productivity—source lines of code.
- Software reuse—lines of code that are reused.
- Software complexity—McCabe cyclomatic complexity.
- Software quality—AdaMAT metrics.
- Software failures—fault-induced software change requests.
- Software reliability estimation and prediction—SMERF is used to predict reliability.
- Development environment resource utilization—on-line storage, CPU time, and input/output.

5.3. The UNAS Project

TRW has used several software development models during the past 30–40 years: the waterfall model, the spiral model, and a newly developed evolutionary model. The latter works by first "pinning down" the software framework, and then developing the application. Framework

---

[7] William H. Farr, Oliver D. Smith, and Carol L. Schimmelpfenneg, "A PC Tool for Software Reliability Measurement," Proc. Institute of Env. Sci. (1988), 271–275.

topics include: operating system, DBMS, communication systems, man–machine interface systems, and similar support software. This framework is developed early in the life cycle and then goes into a maintenance mode. Once the framework is finished, the software for the application is developed on top of it.

The Universal Network Architecture Services (UNAS) product is the outcome of nearly ten years of research and development. UNAS uses TRW's Evolutionary Development Paradigm (EDP), which is based on a modified version of the spiral model developed by Barry Boehm. This development paradigm is targeted toward the development of reliable distributed systems. The fundamental thrusts of the new paradigm are:

- Early identification of the critical system components,
- Rapid development of the first operational prototype (during the first cycle),
- Extensive use of architectural middleware (reusable, black-box software objects).

TRW has been developing the EDP approach over the last ten years and is currently in the fourth-generation product. In generation one, TRW focused on proof of concept and function. Generation two addressed performance. The third generation expanded the architectural diversity to open systems, and the current fourth generation is the UNAS product.

The general model for EDP is shown in Figure 5-1. Many of the activities that must take place during software development are fitted to this model. For example, required assets and attributes are shown in Figure 5-2.

TRW believes that most of the complexity of a computer system (and hence most of the faults) resides in the support software (communications, scheduling, database access, control, and the like), not in the software that directly implements the application functions. Most of this support software is relatively independent of the application, so it can be developed and tested early in the project life cycle, and reused among different projects. This is the approach used in UNAS.

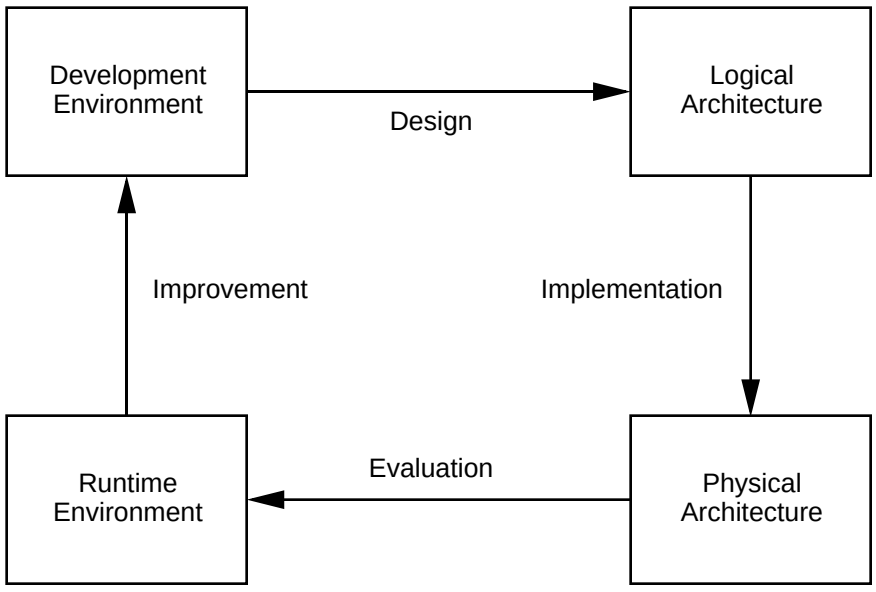


Figure 5-1. The Evolutionary Development Environment and Product

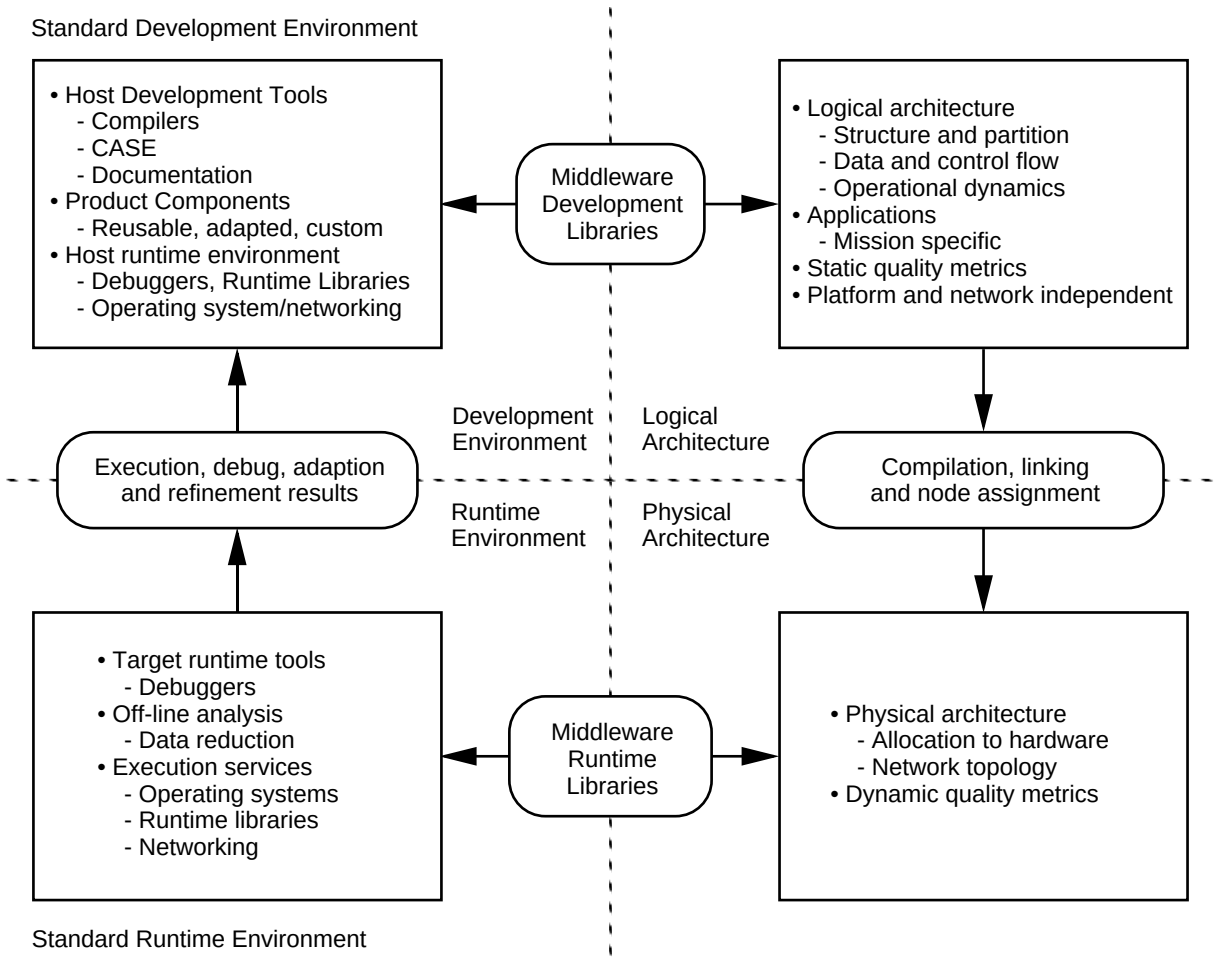


Figure 5-2. Required Assets and Attributes

The structure of a software system can be divided into three layers. The most critical is the architecture, which should be designed by a small group of expert design engineers. The middle layer is devoted to "building blocks" which contain the difficult support software; this should be designed and coded by a somewhat larger group of expert programmers. The actual application is the top layer; it is sufficiently straightforward to permit implementation by ordinary programmers. Figure 5-3 illustrates this concept. This is an example of a hypothetical system, showing the increasing numbers of people, software objects, and interconnections as one moves "up" the layers.

The focus of UNAS is on the "hard, critical" building blocks contained in the center layer, referred to as "middleware." The TRW concept centers on the notion of the reusable software "integrated circuit" (software IC or software component) and the software "circuit board." Each of these conceptual ideas encapsulates the critical functions and thoroughly tests these objects. Typically, the software IC is about ten to fifty source lines of Ada code. The software circuit board is composed of previously tested software ICs and is on the order of 500 to 1000 source lines of Ada code. This concept is similar to the object-oriented concepts of collaborating objects (circuit board) and classes or instances (software IC).

The EDP life cycle approach concentrates on designing and developing the lower-level, more difficult objects early. They are then tested thoroughly, and placed in a maintenance phase for the rest of the development life cycle. This provides the programmers building the application components considerable assurance of the correctness of each component.

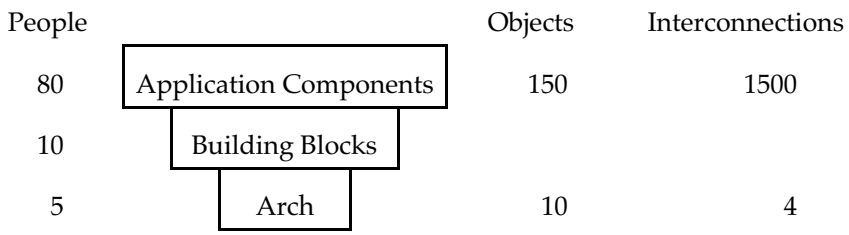


Figure 5-3. Software System Layers

TRW implemented a metrics program for the Evolutionary Development Paradigm. They found that each cycle through the modified spiral model increases the trust in software. Moreover, since the critical components evolve during the first cycle of the project, all additional software is built on top of and exercises these critical components, thereby "continually stressing and testing" them. The first two cycles of the EDP life cycle focus on the system reliability, safety, and critical performance issues.

TRW believes that using the EDP approach and the UNAS in actual projects results in a scrap and rework metric of approximately 25%, compared to the industry norm of approximately 40%. This very significant reduction can be credited to EDP and its "tools."

An example is the CCPDS-R application. This program consisted of three subsystems. Large productivity gains were experienced from the first subsystem to the second. The first subsystem contained ~300K SLOC and was completed in 38 labor months. The second subsystem contained ~450K SLOC and was completed in 20 labor months. This increased productivity is credited to 40% reuse. The final system, containing about one million lines of Ada code, was delivered within time and budget targets.

Productivity gains imply increased "quality" since quality is infused into the product by getting the system into the hands of the users as soon as possible. Additional points are:

- The sooner unknowns can be eliminated or discovered, the more the software quality can be improved.
- Each cycle of the spiral improves quality.
- No first-generation system exhibits excellent quality.
- Reuse folds in lessons learned and helps the spiral process converge to produce a useful product quickly.
- Quality systems are typically built by a small number of people.
- Communication breakdown increases as the size of the project increases.

The following are key factors in TRW's EDP:

- Critical components are built in the first cycle.
- The application builds on these components in all subsequent cycles.
- Program components typically fail at their interfaces, so EDP has moved integration into the design phase.
- EDP makes very early, immature models of the product and evolves them to the final product.
- The product and initial model intentionally differ in level of maturity.
- Reuse of proven middleware is critical, since 80% of software problems come from building onto operating system services (software ICs and software circuit boards).

Multiple metrics are used in UNAS application development for monthly program management reviews. These provide different perspectives on development progress by using a variety of indicators. The metrics are:

Staffing profile	Staff attrition and addition

Software complexity	Software size
Software progress	Cost and schedule variances
Software architecture stability	Program volatility
Software Problem Report volume	Scrap and rework
Action item closure progress	Software Problem Report closure progress
Host target resource utilization	Documentation progress
Test progress	Requirements verification progress

UNAS is discussed at length in an article by Royce and Royce.[8] It is interesting to note that TRW now provides warranties with code developed under UNAS. This is not because the software is error-free, but because UNAS makes the software cheap to fix.

The benefits of UNAS to TRW are summarized in a TRW table reproduced in Figure 5-4.

5.4. Discussion

Discussions with TRW personnel revolved around two key questions. First, what are the most important obstacles to producing highly reliable software for use in safety-related applications? Second, why is TRW successful at producing highly reliable software? In each case, certain hypotheses were posed to TRW for consideration. These hypotheses, and TRWs response, are discussed below.

5.4.1. Obstacles to Producing Highly Reliable Software

The following list contains eight obstacles to producing highly reliable software, based on information received from TRW. The contents of this list show two patterns. First, both psychological and technical concepts are involved, and some items have aspects of both. Second, some items appear fundamental and others less so. The fundamental questions are probably more important, and harder to solve. The list gives the proposed obstacle followed by TRW's response.

- Obstacle: Inaccurate interpersonal communication.

Response: Dead right. UNAS is an attempt to solve this by specific provisions to resolve communication problems in meetings and in high-level documentation. It was noted that this statement is somewhat controversial within TRW. The need for tools to enforce compliance with standards was emphasized. Tools are impersonal, and help eliminate the human ability to ignore orders.

- Obstacle: Lack of mathematically based "irrefutable first principles" for software design.

Response: This principle affects the whole approach to software engineering, to writing reports, to creating designs, and so forth. What is the answer? One possibility is experimenting—which is an argument for prototyping.

- Obstacle: The discrete (non-continuous) nature of logic, upon which software is based.

Response: Yes.

- Obstacle: Inability to precisely translate informal requirements (user desires) into formal specifications.

Response: The conceptual design review (CDR) is used for mid-course corrections. It is possible to use tools, such as screen generators, to obtain customer and developer agreement on requirements. The problem can be solved, but that solution requires planning and careful work by both the developer and the customer.

- Obstacle: The size and complexity of the applications.

Response: This is true, but there were no suggestions on how to solve the problem. This may be one of the most difficult problems for safety-critical software.

[8] Walker E. Royce and Winston W. Royce, "Software Architecture: Integrating Process and Technology," *Quest* (Summer 1991), 3–16.

Quality Impact	Conventional	UNAS
Requirements misunderstanding	Discovered late	Resolved early
Development risk	Unknown until late in process	Resolved early
Reusable software	Not supported	A natural attribute
Change management	Chaotic and malignant	Straightforward and benign
Design errors	Discovered late	Resolved early
Warranted performance	Unheard of	Practical and feasible
Cost estimation	Unpredictable	Predictable
Schedules	Incompressible and protracted	Tunable to quality, performance, function and technology
Target resource estimation	Qualitative, simulation, analysis	Quantitative, executing prototypes

Figure 5-4. Quality Improvements with UNAS

- Obstacle: The need for ultra-precise thinking.

Response: Yes.

- Obstacle: Requirements instability.

Response: This is a big problem. It is especially important in safety-critical systems—it is really necessary to have a frozen set of requirements in such cases.

5.4.2. Success in Highly Reliable Software Production

The next list contains seven suggested design factors which partially explain why TRW has been successful at producing software with very high reliability. The list gives the suggested factor followed by TRW's response. TRW gave three additional factors, which are provided in the paragraph following the list.

- Factor: TRW has been creating highly reliable systems for 30–40 years, and has a great deal of experience in doing so.

Response: Yes. TRW has had some failures. There is a big variation in the safety-importance of current projects. Different groups within TRW are at different levels of the SEI maturity model. The following table shows the approximate percentage of projects at different levels. TRW has a corporate goal to move all projects to level three; they expect to have about 75% of their projects at level three within the next few years.

Level	Percentage of Projects
1	50%
2	10–15%
3	20–30%
4	2%
5	0%

- Factor: TRW has some highly qualified people, who have been thinking about software development for many years. An example is the ten-year development effort seen in UNAS.

Response: True. TRW employees have invented new process models three times. An example is Barry Boehm's highly successful spiral model of the software development process.

- Factor: There is a management commitment to quality. An example of this is the Software Engineering Process Groups.

Response: Yes.

- Factor: TRW is prepared to experiment with new methods (such as the EDP modification to the spiral life cycle), to improve methods that work and to abandon methods that do not.

Response: Yes, sometimes in an organized way; sometimes, not. In either case, individuals matter—people taking lessons learned from one project and applying them to the next project. It does not work by managers giving orders, but management support is important. Professional people do this instinctively.

- Factor: TRW understands the importance of folding "lessons learned" into new projects.

Response: True, as mentioned in the previous answer. LLNL asked if TRW is a good example of the NASA/SEL Experience Factory; the answer was "yes," but there is no direct comparison to SEL.

- Factor: TRW believes it has identified some major impediments to creating reliable software, and has taken steps (in UNAS) to control the impact of such impediments. The most important of these is communication within a large development team.

Response: Yes. The importance of UNAS was summarized as (1) getting rid of race conditions and deadlocks by guaranteeing that they cannot occur, (2) automatic coding aspects, and (3) communication.

- Factor: A layer of software has been identified which is both difficult to develop, and has a high potential for reuse. By a deliberate concentration of this layer, those portions of software which have a high potential to fail can be identified, developed carefully, and tested thoroughly. This results in highly reliable modules, and (as a consequence) highly reliable application systems.

Response: Yes.

The following points were added during the discussion. TRW is successful because (1) it is a software company, (2) it attracts good software people, and (3) software is the primary business. This latter point contrasts TRW with, for example, an aircraft company, where software development is a secondary business.

6. AIAA SOFTWARE RELIABILITY WORKING GROUP

6.1. Background

The American Institute of Aeronautics and Astronautics (AIAA) Space Based Observation Systems Committee on Standards (SBOS COS) created a working group on software reliability in August 1989. This working group was created "to study issues in the area of software reliability measurement."

The working group charter is "to standardize methods for the assessment of risk and prediction of software failure rates." Four major projects were defined:

- A national repository for software reliability data,
- A software reliability measurement tool survey,
- A software reliability recommended practice,
- A set of software reliability "best current practice" documents.

The first three projects are described briefly here.

6.2. AIAA Software Reliability Database

The Software Reliability Database is intended to contain reliability data on "numerous completed software development programs."[9] The intent is to create a national repository of reliability data for actual software systems. The data will be organized by project, and will contain some of the following information:

Project data.

Name of each life cycle phase.

Start and end date for each life cycle phase.

[9] David M. Siefert and George E. Stark, "Software Reliability Handbook: Achieving Reliable Software," Third Int'l Symp. on Soft. Rel. Eng. (Oct. 7–10, 1992), 126–130.

Effort expended, in staff months, for each life cycle phase.

Average development team experience.

Tools and methods used for requirements, design, code, test, and CM.

Number of organizations involved in the project.

Constructive cost model (COCOMO) development environment.

Software Engineering Institute index of the development environment.

Tool and model used for software reliability estimation.

Component data for each component.

Software size in LOC, number of comments, number of object instructions.

Source language.

Name and model of development and target hardware.

Average and peak computer resource utilization.

Dynamic failure data for each failure.

Life cycle phase during which the problem was detected.

Date and time of failure.

Number of CPU hours since the last failure.

Number of test runs or test cases executed since the last failure.

Clock time since the last failure.

Test hours per test interval, and number of failures detected in the interval.

Test labor hours since the last failure.

Severity of the failure.

Type of failure.

Method of failure detection.

Unit complexity of failed unit.

Fault correction data for each corrected fault.

Date and time the fix was available.

CPU hours required for the fix.

Number of runs required to make the fix.

Clock time used to make the correction.

Labor hours required to make the correction.

The database currently includes some data sets from Lockheed, IBM Federal Systems Company, NASA Johnson Space Center, and Bendix Corporation. The majority of the data is from DoD, FAA, and NASA projects. The data that was submitted to the Working Group is presently in "raw form;" there has been little, if any, analysis of it. The Working Group considers that disassociating the data received from the source of the data is of paramount importance. The database schemata to store and retrieve this data is still being designed. The current design goals include using a commercial-off-the-shelf database management system running on a PC platform.

6.3. Software Reliability Tools Survey

A survey of software reliability measurement tools was undertaken and documented.[10] Four tools were investigated in some detail: SMERFS, SRMP, GOEL and MUSA. The Statistical Modeling and Estimation of Reliability Functions for Software (SMERFS) was judged the most comprehensive, readily available, and machine-independent. PC-based software tools tend to be upgraded quite frequently, so this judgment has limited utility.

6.4. Software Reliability Models

Fifteen software reliability models were investigated, classified, and evaluated. Selection criteria were as follows:

- Model validity.
- Ease of parameter measurement and interpretation.
- Quality assumptions.
- Applicability to different development methodologies.
- Simplicity.
- Sensitivity to "noise" and input data parameters.

[10] George E. Stark, "A Survey of Software Reliability Measurement Tools," Int'l Symp. Soft. Rel. Eng. (1991), 90–97.

Two major categories were used to classify the models, one of which has three subcategories. These categories, with the recommended models for each, are as follows:

- Error count models. These models use test intervals and the number of failures in the test intervals as input data. The Schneidewind model is preferred.
- Time domain models. These models use time between failures as input data. There are three subcategories:
  - Exponential time domain models. The Jelinski-Moranda model is recommended.
  - Non-exponential time domain models. The Musa-Okumoto Logarithmic Poisson model is recommended.
  - Baysian time domain models. The Littlewood-Verrall model is recommended.

6.5. AIAA Software Reliability Recommended Practices

The main accomplishment of the AIAA SBOS COS has been to write and publish a recommended practice for software reliability.[11] The purpose of this handbook is to recommend quantitative tools and procedures for estimating the reliability of a software system. The recommended practice can be used from "the start of the integration test phase through the operational use phase of the software life cycle."

Six topics are discussed in the recommended practice:

1. Common terminology.
2. Software reliability estimation procedures. This section describes an eleven-step generic procedure which can be followed (and tailored) to perform an analysis. The steps are:
   a. Identify the application.
   b. Specify the requirement.
   c. Allocate the requirement.
   d. Define "failure."
   e. Characterize the operational environment.
   f. Select tests.
   g. Select models.
   h. Collect data.
   i. Determine parameters.
   j. Validate the model.
   k. Perform the analysis.
3. Model selection. This section describes each model's assumptions, objectives, data requirements, reliability mathematics, implementation status, and reference applications. Criteria are provided for selecting a model. The models mentioned in Section 7.4 are discussed in some detail.
4. Data collection. Data must be collected during late life cycle phases (integration, testing, operations) for use in analyzing the product and fine-tuning the model that is used.
5. Open research questions.
6. Predicting system failure rates.

[11] Recommended Practice for Software Reliability, ANSI/AIAA R-013-1992.

7. UNIVERSITY OF MARYLAND AND NASA SOFTWARE ENGINEERING LABORATORY

This section documents a conversation held with Victor Basili, University of Maryland. Highlights of the conversation, as they relate to the subject of this report, are given below.

7.1. Document Reading

Human analysis of documents (including computer codes) is an area with a high potential payoff. The software engineering community does not yet know how to effectively and efficiently read and analyze documents, and it needs to learn to do this. Software engineers do generally understand how to organize the reading process, through design reviews, but understanding the organization of reviews is not the same as understanding the subject matter of such reviews.

The clean room technique was a start at this, but it did not go deep enough. A reading technology

needs to be invented. How should a requirements specification be read? A design specification? A body of code?

The NASA Software Engineering Laboratory (SEL) has run some small experiments on reading. Some elementary tools based on checklists were constructed. The checklists were based on different classes of errors: ambiguities, inconsistencies, omissions, incorrect facts, material in the wrong place in the document. The method requires several readers, each of whom looks for errors of a particular type. At present this is a manual approach, but machine assistance may be added at a later time.

The result of the first experiments with this method was very positive. Having individuals look for particular types of errors was clearly superior to having everyone use the same checklist and look for all types of errors. That is, by focusing one's effort on a particular type of error, one increases the number of errors found of that particular type.

Another approach might be phase-based reading. This has been considered, but no experiments have been run. In this approach, the set of readers will include people of different backgrounds, looking for errors in the document that affect their individual interests. A tester, for example, will read a specification with the goal of being able to create a test of the specification. An application expert will check that the specification accurately reflects the application—for example, that no physical laws are violated. Maintenance personnel will examine the specification to see if the resulting product will be easy to maintain. A safety expert will look for risks. This approach to reading can be termed "reading selfishly."

7.2. Life Cycle Comments

Each phase of a life cycle can be thought of as containing two parts: a construction part and an analysis part. If software engineers can learn how to do the analysis effectively, they should be able to obtain valuable insight into how to do the construction properly.

Analysis can be error-based or phase-based. Figure 7-1 shows the analysis activity that occurs after the requirements specification has been constructed; a similar activity would occur after all other life cycle phases. The diagram shows two types of analysis taking place on the specification.

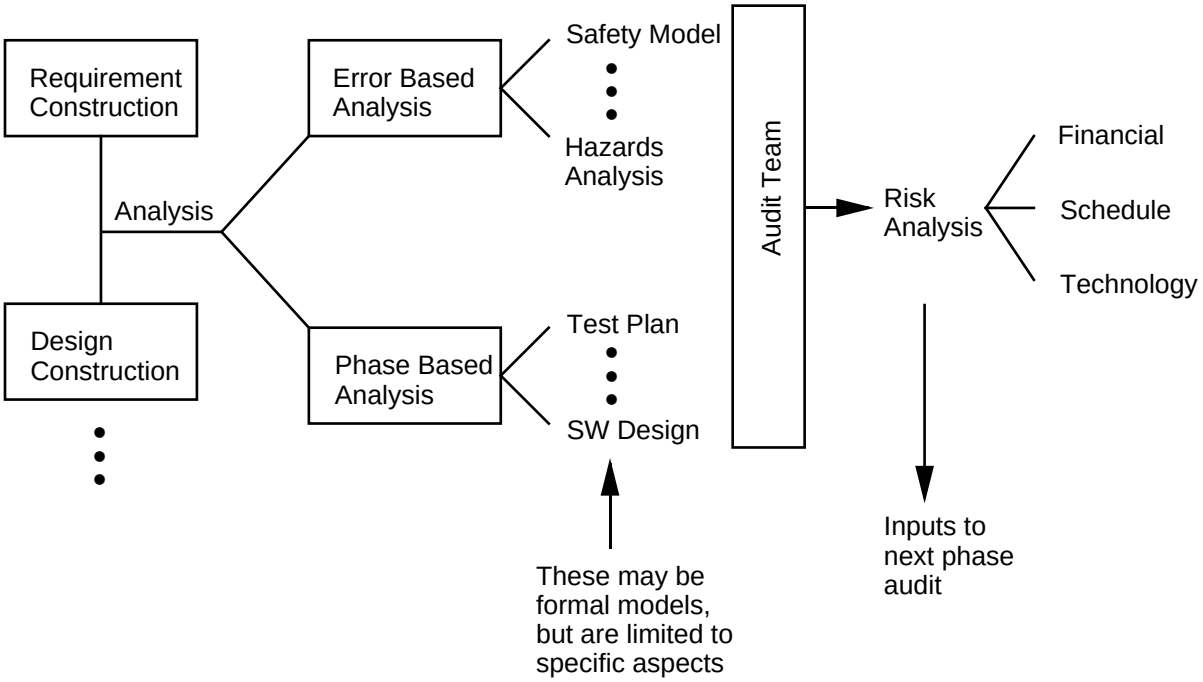


Figure 7-1. Life Cycle Phase and Construction Analysis Activities

These analyses may be of any type; in the example, it is assumed that models will be developed. The first type of analysis looks for errors, using the ideas discussed in Section 7.1. The other form of analysis is based on the specific life cycle phase under consideration. For example, an initial test plan can be written, and an initial attempt at a software design can be produced. If these can be done in a reasonable manner, confidence in the requirements specification is increased.

An audit team will examine the specification and the various analyses to be sure they are done correctly. One set of products of the audit will be risk analyses. Three are suggested here: financial, schedule, and technology. These risk analyses will be kept and used as input to the audit of the next phase; that audit will pay special attention to how the risks were handled by the construction team. In this example, suppose that the audit of the requirements specification determines that there is a significant risk that the schedule cannot be met. This will be documented. After the design specification has been completed and analyzed, the design audit team will want to know what was done to address this schedule risk. An additional risk analysis will be done on the design; perhaps the design is complex, requiring new workstation technology. This will be documented, and used as input to the next phase audit, and so on.

7.3. Miscellaneous Comments

It is possible to measure a process. For example, a tester can be asked, "Do you understand the document well enough to write test cases?" The answer will be "yes" or "no," and this is a measure. Another possibility is to ask for a value in a range—say, 1–5. The meanings of the numbers must be defined carefully and used precisely. This approach is taken quite often in social science surveys, and is apparently effective there. Measuring is important from a psychological standpoint, since it tells the developer that his opinion is important.

The members of an audit team should be selected carefully according to their areas of expertise. The audit team might, for example, include a software engineer, a tester, a domain expert, a quality assurance person, a configuration manager, and a safety engineer, permitting the various audit team members to look primarily for problems in their own areas of expertise. With a properly selected team, all areas of importance should be able to be covered.

The optimal size for modules, when measured against defect rate, is application-dependent, but is probably about 300–400 lines of code. Figure 7-2 illustrates this concept.

Three things are necessary to achieve safety:

- A well-articulated set of goals.
- A well-thought-out set of processes that are likely to achieve the goals.
- An assessment system (measurement) that can provide assurance that the process has achieved the goals.

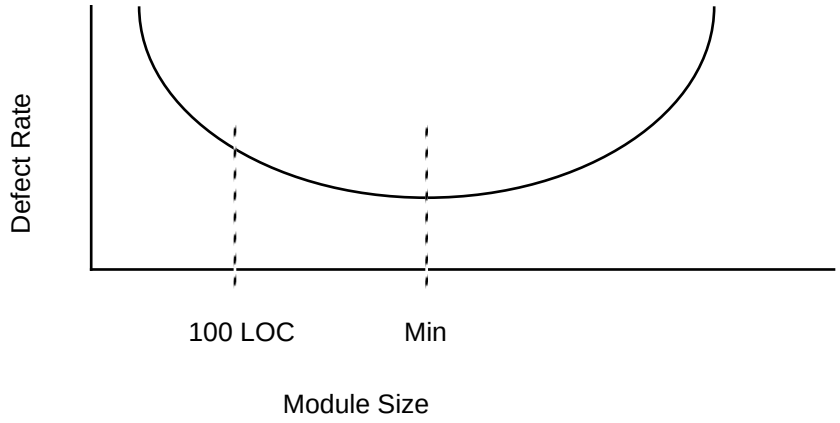


Figure 7-2. Estimated Optimal Size for Modules